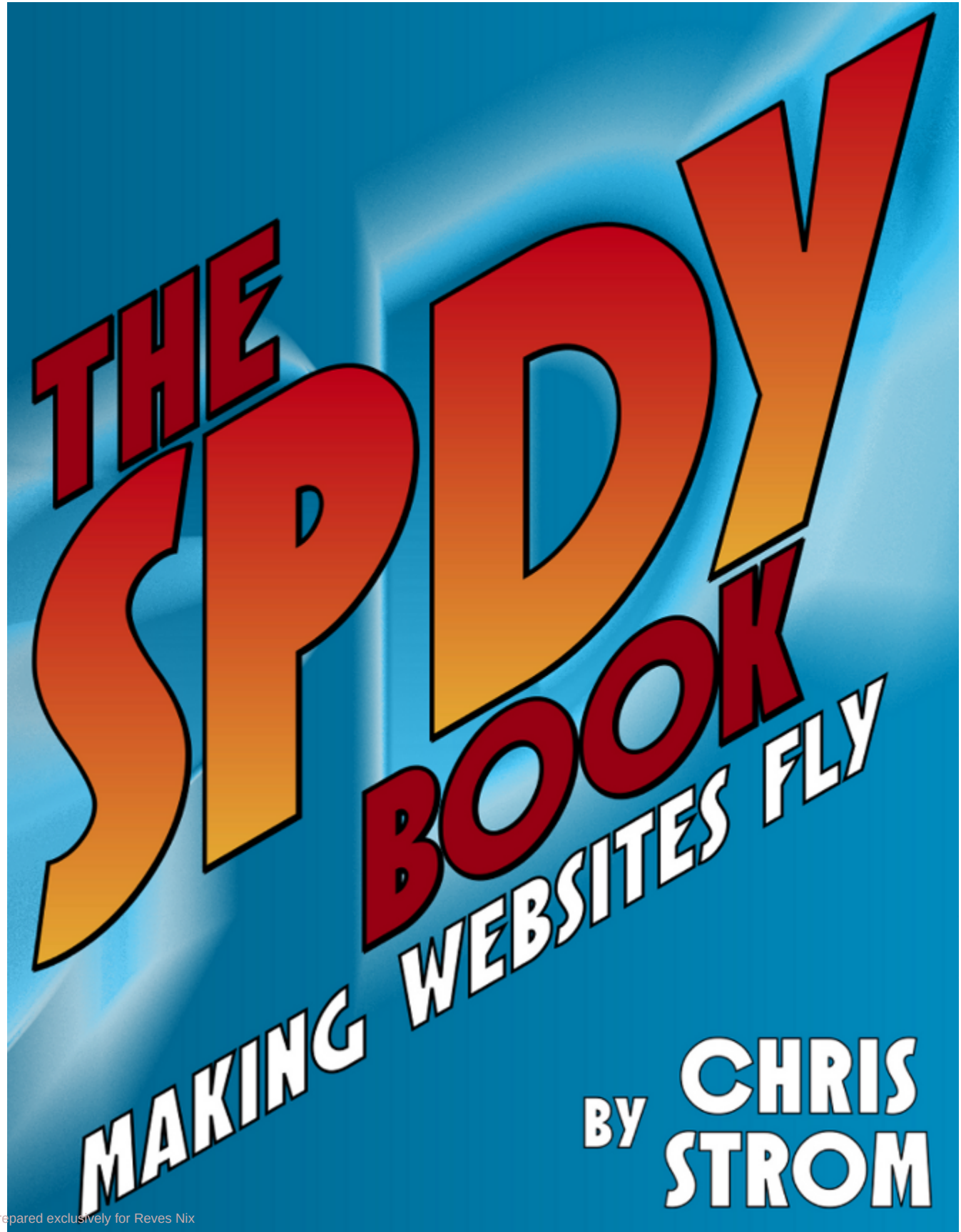


# The SPDY Book

**Chris Strom**

# The SPDY Book

Chris Strom

# Table of Contents

# Copyright



If you didn’t pay for this book, please do the right thing and compensate me for my time at spdybook.com. If you don’t want to fork over a bit of cash, nearly all of my notes are available for free:

http://japhr.blogspot.com/2011/04/my-chain-3.html

But seriously, do the right thing.

# History

- 2011-06-30: First beta release.
- 2011-07-05: Formatting changes. Copy edits from Ashish Dixit.
- 2011-07-19: New application-centric chapters at the beginning of the book. Thanks to Dave Thomas for the suggestion. New chapter on control frames. Incorporate many suggestions from Mike Belshe.
- 2011-07-28: Last two chapters.
- 2011-07-29: More alternatives chapter work. Suggestions from Jeff Dallien, Mike Belshe and Gautam Dewan.
- 2011-07-30: Finally write the wireshark appendix. Corrections from Jeff Dallien.
- 2011-07-31: Final release. Add example of what is possible for future use. Many, many edits.

# Dedications and Acknowledgments

How could this book be dedicated to anyone besides my wife? Not only did she put up with 99 straight days of my obsession, she helped every step of the way. Quite beyond just being patient, she gave of her time and even served as the primary editor of this book. Robin, I don't know how you do it all, but I love you!

Huge thanks to all of my beta readers. Even the smallest copy suggestions were of great help in helping the overall quality of the book. Special thanks to Jeff Dallien, Gautam Dewan and Ashish Dixit for multiple suggestions. How you guys find the time to provide such thoughtful, helpful comments, I do not know. But I'm grateful that you do!

I am quite grateful to Dave Thomas for his comments on the overall structure of the book. The *SPDY Book* would have been much poorer for it if he had not pointed me in the right direction.

Special thanks to Mike Belshe for technical insight into the book and his sometimes nightly guidance on my blog. And, of course, thanks for helping to write the SPDY spec in the first place! I can only hope that this book adds to the great work that you have already done.

# Introduction

HTTP is an astonishing accomplishment.

Originally created 20 years ago and last updated more than 10 years ago, it is hard to understate what has been accomplished with it.

Conceived and solidified before iTunes and iPads, before Android and before Facebook, it has nevertheless allowed those things to thrive.

SPDY is not meant to stave off some imminent collapse. HTTP and browsers are not in dire shape.

But there is still waste. There is still room for improvement. There are many, many lessons learned in the past 10 years.

Google has made a first attempt to describe what a protocol built for the future might look like. At the time of this writing, SPDY is not even an IETF [1] recommendation.

It is barely a gleam in its parents' collective eye. And yet it is still important. Possibly vitally important to the future.

It is called SPDY and its name is meant to evoke its purpose: PAGE LOAD SPEED.

After reading this book, I hope that you will:

- understand the problems that SPDY is trying to solve
- be fluent in SPDY sessions, including reading individual packets
- SPDY-ize your own sites—either by writing your own SPDY parser or using one of the frameworks discussed.

[1] Internet Engineering Task Force - http://www.ietf.org/

# 1. Who Should Read This Book?

This book is primarily intended for web developers, mobile application developers and network administrators.

It is ideal for anyone wanting to make their sites faster. Even if you decide that you do not want to use SPDY, there ought to be oodles of tidbits that you can apply to your existing HTTP/1.1 sites.

SPDY is built on top of HTTP. It is meant to transport web traffic. As such, good HTTP fundamentals will help when reading this book. Even so, you need not be an expert on all things RFC 2616 [2]. Wherever possible, I try to explain the HTTP involved—especially as it relates to SPDY.

By profession, I am a Ruby and Node.js hacker. As such, I will focus my discussion on the SPDY gem from Ruby and the node-spdy module from node-land.

Even if you have little-to-no experience with Ruby, I try to keep the Ruby code samples very readable (one of Ruby's hallmarks). I do assume some experience with Javascript—especially in later chapters.

# 2. Too Early?

Given that SPDY has yet to be submitted as a recommendation to any standards body, it can be legitimately asked: is a book such as this too early?

Maybe, but I would argue not. Well duh, I wrote the damn thing, but here's why…

The problems that SPDY tries to solve are not going to magically go away.

Even though SPDY is not yet a formal recommendation, it has gone through two drafts. A third draft is underway at the time of this writing. After the third draft is complete, SPDY will be submitted to the IETF. Bottom line: this may be the perfect time for a book!

[2]The HTTP specification - http://www.ietf.org/rfc/rfc2616.txt

Regardless, I am *not* writing the definitive guide to SPDY. It is my hope that this will be a solid grounding in SPDY fundamentals and no more.

# 3. Can You Afford to Ignore SPDY?

33% of broadband users would rather close the browser tab loading your site than wait 4+ seconds [3]. You need to be doing everything in your control to ensure that your site is *reliably* available to users all over the world in less than 4 seconds.

You could spend time and resources implementing numerous suggestions for page speed [4]. Or you can embrace a technology supported by the fastest growing browser on the market (Google's Chrome).

SPDY comes with the promise of reducing page load time by 50%. If your site takes 8 seconds, you can almost immediately realize the goal of a 4 second site. If your site takes less time, you can make your site even faster or *add* cool new features that you wouldn't dare add today.

SPDY promises faster sites, but also more reliable sites. SPDY works with the underlying TCP/IP networking layer instead of abusing it like HTTP does [5].

Currently, SPDY is only available for the Chrome browser, but how much longer can other browser vendors ignore SPDY if it makes things 50% faster?

How long can you afford to ignore it?

[3] http://www.akamai.com/dl/reports/Site_Abandonment_Final_Report.pdf
[4] http://code.google.com/speed/page-speed/
[5] More on HTTP's abuse of TCP/IP in Chapter 1, *A Case For SPDY*

# Chapter 1. A Case For SPDY

In the Beginning…

…the web was simple. Web sites were simple. Web pages were simple.

Sites had few pages. There was little interactivity.

There were one or two images. All styling took place within the HTML itself. Simple…


And HTTP/1.0 was plenty.

Then the web grew. Users needed more and sites wanted to provide more. Pages got larger. Javascript was introduced to provide dynamic client-side interaction. Styles were split out into CSS stylesheets. More images were added.


And HTTP/1.1 was plenty.

But that was 10 years ago.

It is remarkable that the web has evolved as it has all on the back of HTTP/1.1. We have seen the rise of Google, Amazon, Facebook and Twitter. All on the back of HTTP/1.1.

Pages nowadays are far more complex than they were even a few years back. The typical page has 70+ additional requests [1]. After the page itself, there are dozens of images that

[1] http://httparchive.org/trends.php?s=intersection

need to be loaded. Javascript, once loathed by developers, is now critical for even the simplest sites. Not only is Javascript required on the page, but Javascript frameworks and libraries are increasing in size and number. We need to load in jQuery core and jQuery UI. We need to add backbone.js and underscore.js. And stylesheets are growing in complexity, size and number.

In the future, pages and sites are not going to get any simpler.

Beyond simple evolution, what might the web look like if it were easy to load hundreds of resources along with the homepage? What cool things are we not thinking about? Dashboards? Games? Collaboration tools?

SPDY is a new protocol, built on top of HTTP/1.1, that allows us to answer those questions. And more.

# 1.1. Bandwidth Will Save Us… Right?

Most web developers hold in the back of their minds the idea that all of the new and awesome stuff just starting to percolate will be achievable in a couple of years. Achievable thanks to increased bandwidth.

In 2012 we are just embracing a 4G world. Soon we expect that 10mbits/sec will become common place. In a few years, nearly everyone will double, triple, or increase that tenfold. And then just imagine what browsers can download and kickoff.

Right?

Well no. If that were the case, then there would be no need for SPDY.

The problem is that, due to fundamental limitations of a synchronous protocol like HTTP sitting on top of an asynchronous protocol like TCP/IP, we are reaching the point of diminishing returns in website bandwidth.

It turns out that page load time for a moderately complex site is almost no faster over 20mbits/sec than over 2mbits/sec.

We will discuss more about the reasons for this in the next chapter, but this is what SPDY will help with.

# 1.2. Why Not HTTP/2.0?

Someday, if SPDY continues to be successful (hey it's already got its first book, right?), it may be the basis for HTTP/2.0.

For now, it is content to be a start. It is a good start, drawing on the lessons learned from the past 10 years of life with HTTP/1.1. None of the ideas are particularly new. All of them—compression, multiplexing, prioritization—have been discussed, suggested and tried before. SPDY, however, is the first attempt to include all of those ideas in one place.

Rather than write the perfect specification and wait for everyone to get excited by it enough to begin implementation work, the authors took the opposite approach. They started with the implementation and tested it like crazy. Not content to test it in the lab, they tested in the real world—on Google's production servers with real users (and lots of them).

For the immediate future, the SPDY authors have come up with a way to co-exist with existing HTTP applications.

# 1.3. Is Google Trying to Embrace and Extend™?

Google web browsers run *significantly* faster with Google web sites [2].

A few years back, if another company did something similar, there would have been an uproar that said company was attempting to usurp the web for its own nefarious purposes and lock out other vendors in the process.

So how can Google claim that it is not Embracing and Extending™ the web?

The answer is simple: everything is public.

[2]Google would, of course, claim that Google browsers run faster against all sites ;-)

They are publishing and documenting everything publicly. They collaborate with the public. They encourage and help authors attempting to port SPDY to alternate programming languages.

They maintain a mailing list (spdy-dev) on which they genuinely engage the community and take feedback to heart. There have been numerous features of the first 2 drafts that have been dropped from subsequent specs based on feedback and discussion on spdy-dev.

The largest experimental protocol in daily use is… completely public.

But enough talk, let's see some of this in action…

# Chapter 2. Your First SPDY App

I’m not gonna mess around in this book, so let’s jump right into a SPDY `hello world` app:

```
app.get('/hello', function(req, res){
  res.render('hello', {
    title: 'Hello World!'
  });
});
```

If you are familiar with node.js, you will likely recognize that bit of code.

If you have never done node.js, that code is still pretty easy to follow thanks to a simple DSL (Domain Specific Language). This DSL describes how to route an HTTP `GET` request for the `/hello` resource. In this case, the response merely renders the `hello` template, passing along a `title` parameter of "Hello World".

Now you might be thinking to yourself, "I plunked down good money to learn SPDY and he’s talking about serving up HTTP with node.js?! How do I get a refund?" First off, all sales are final—no refunds. Just kidding.

Actually, this is lesson #1:

**Tip**

*Lesson #1:* SPDY serves up HTTP applications. As we will see, it definitely changes a few things here or there. The SPDY protocol builds on HTTP, it does not replace it.

So is that it? Nothing else is needed to run a SPDY application?

## 2.1. OK I Lied

The `hello world` sample app is not *really* SPDY. Yet.

The DSL that the sample app uses comes from the express.js framework. If you have ever seen the Sinatra framework in Ruby, then you understand what express.js is meant for—a simple way to write HTTP applications.

So what do we need to do to SPDY-ize it?

Two things. We need an adapter layer, `express-spdy` in this case, and some configuration:

```
// Require necessary modules
var express = require('express-spdy')
  , fs = require('fs');

// Create an instance of an application
// server -- SPDY-ized
var app = express.createServer({
  key: fs.readFileSync('keys/spdy-key.pem'),
  cert: fs.readFileSync('keys/spdy-cert.pem'),
  ca: fs.readFileSync('keys/spdy-csr.pem'),
  NPNProtocols: ['spdy/2', 'http/1.1']
}

app.get('/hello', function(req, res){
  res.render('hello', {
    title: 'Hello World!'
  });
});
```

In a regular express.js application, the `express` variable would come from requiring the `express` module. The `express-spdy` module takes the express.js framework and adds a few features to it—kinda like SPDY adds a few things to HTTP.

The API for express-spdy is unchanged from express.js, including the call to `createServer` that returns an express.js application object. Here, we are returning a SPDY-ized version of the express.js application object.

The first three configuration options that are passed to `createServer` — `key`, `cert`, and `ca` — are SSL (Secure Socket Layer) options. In fact, those options come straight out of express.js. They are, of course, not required in express.js, but with SPDY, SSL is **mandatory**. We will cover SSL in depth in Chapter 8, *SPDY and SSL*.

**Tip**

*Lesson #2:* SPDY is served over SSL. In an age where session hijacking is trivial thanks to tools like Firesheep, all non-trivial applications should be served with SSL. There is a performance hit by using SSL, but SPDY acknowledges this as a price of writing serious, modern applications.

The next configuration option, `NPNProtocols`, is the real difference between an express.js app and an express-spdy app. NPN, or Next Protocol Negotiation, is a relatively new feature of SSL. NPN is a vehicle for web servers to communicate to browsers the protocols that they can speak.

In this case, we are advertising that our `hello world` application can speak both `spdy/2` (SPDY, version 2) and `http/1.1`. Browsers that do not understand NPN or SPDY can still make regular HTTP requests of our `hello world` server.

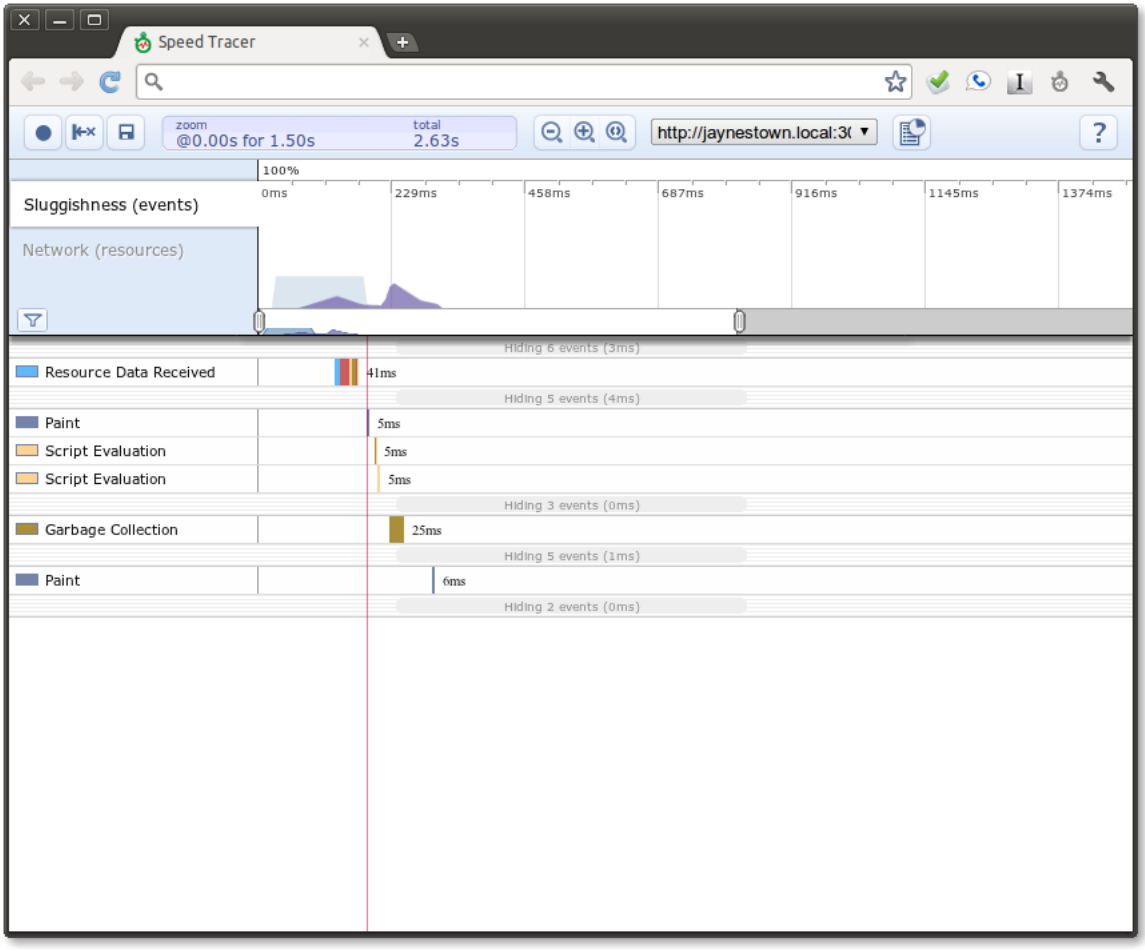
But browsers that understand NPN and SPDY, such as Google's Chrome, will speak SPDY to the server. And it will make a *huge* difference.

Let's find out how…

# 2.2. Without SPDY

As we explore SPDY, we will need to be able to compare it with vanilla HTTP and, later, alternative protocols. The Speed Tracer extension for Chrome produces some very nice graphs for just this purpose.

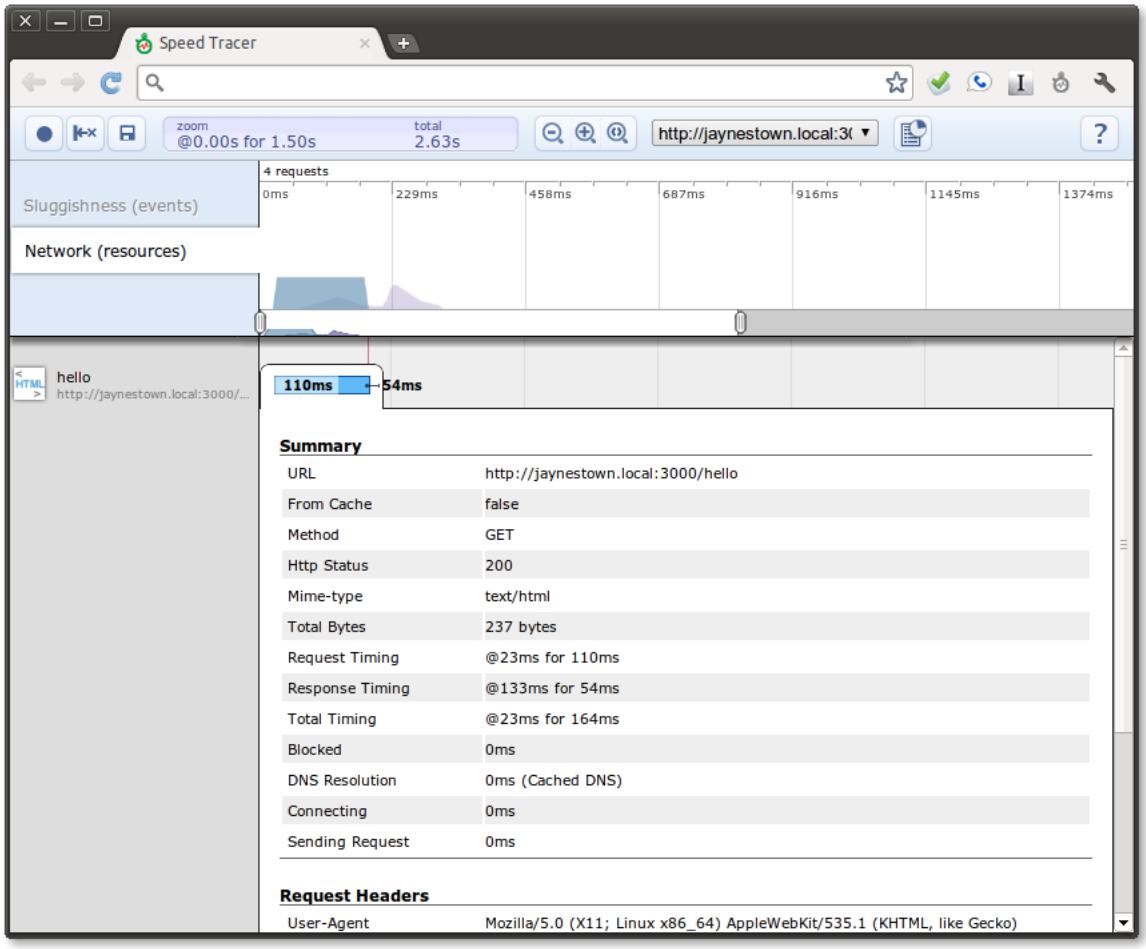
The `hello world` app *without* SPDY looks like this in Speed Tracer:




Speed Tracer presents a nice overview of just about anything that might affect page load time including: network overhead, parsing of CSS/HTML/Javascript, rendering, even internal garbage collection.

From this Speed Tracer overview, we see that most of the time is spent receiving data from the server. It looks as though even more time is spent waiting for the data in the first place —it only takes 41ms to process the data, but there is almost 100ms before data is even seen here. What gives?

The answer to that can be found in the "Network (resources)" tab of Speed Tracer:




The request took 110ms while the response took another 54ms. Why so long? Network latency is causing a delay here. Real world connections are not instantaneous and SPDY is optimized for the real world.

Local connections usually have round trip times of a couple milliseconds. Great connections on the internet are on the order of tens of milliseconds. Round trip times of 100ms are not great, but also not uncommon.

**Note**

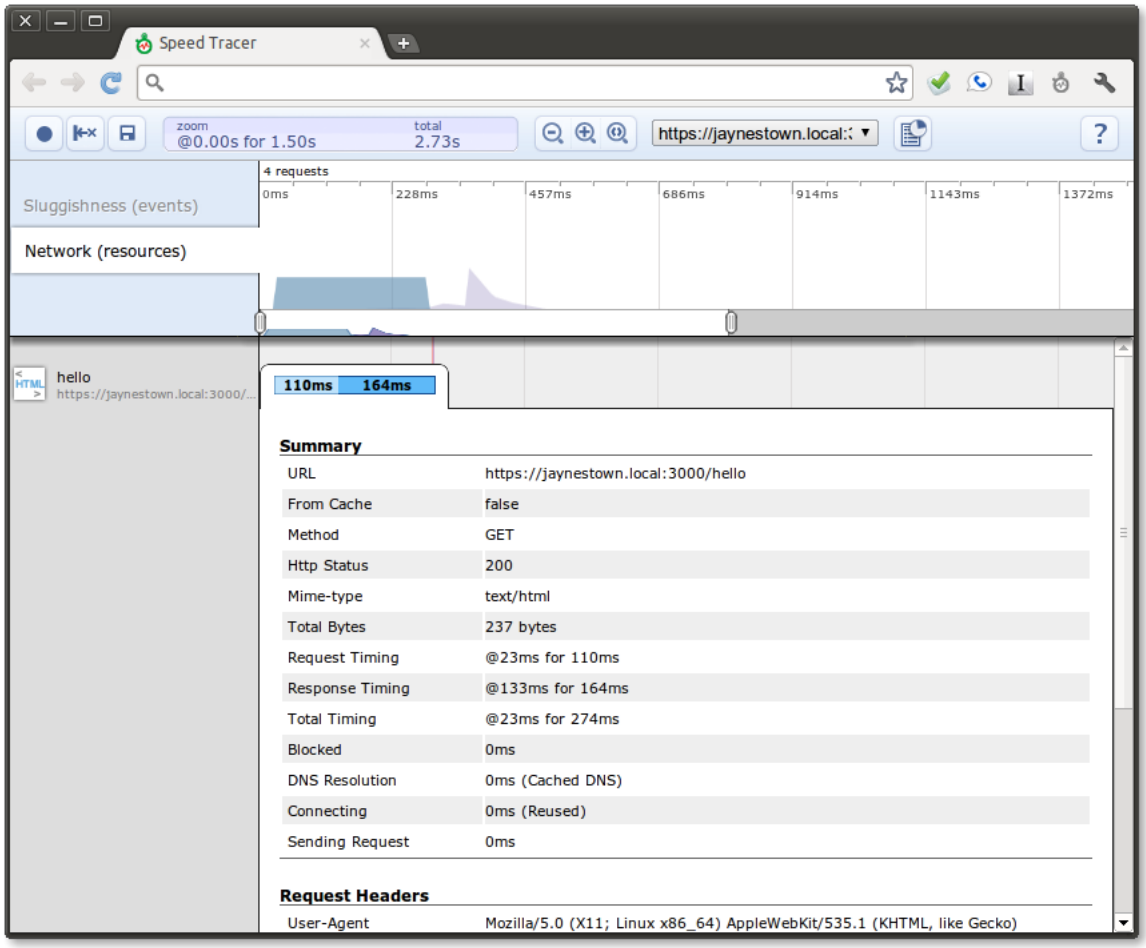
Unless otherwise stated, all examples in this book will include a one-way latency of 50ms / round trip latency of 100ms.

What this tells us is that the biggest problem facing our `hello world` app is network latency.

Surely SPDY can help! Let's see what SPDY is capable of…

# 2.3. The SPDY Difference

After adding `require('express-spdy')` and `NPNProtocols: ['spdy/2', 'http/1.1']` to convert the express.js app to express-spdy, Speed Tracer reports:




Now wait a second. In both the vanilla HTTP version and SPDY version of the application, the request takes 110ms. But the SPDY response takes 133ms vs 54ms in vanilla HTTP. The response time in SPDY is more than double?

No doubt you are really beginning to question the wisdom of reading this book, but have no fear. Speedy is just around the bend.

### Note

The slowness in response time is due entirely to a small penalty paid in the very first frame of a SPDY session. The first frame contains compression initialization

that increases the compressed frame beyond the size of the original. Once we hit the Advanced Topics section, we will see why this is and why the penalty is very, very much worth the price.

Mostly, the slower response time boils down to an infallible truism…

# 2.4. Hello World Applications Are Silly

The hello world application in this book is no exception.

SPDY is overkill for simple sites like this `hello world` application, documentation repositories, or marketing sites. But if you are planning on doing anything more complex than this, SPDY will give you a huge win.

### Tip

*Lesson #3:* SPDY is not optimized for very small applications.

In the next chapter, we will get our first taste for the power of SPDY as we add some images and simple Javascript to our hello world application.

In this chapter, we saw our first SPDY application. Happily, SPDY apps are not that much different than the HTTP apps with which web developers are familiar. To be sure there will be much to learn about writing SPDY apps, but at least we do not have to start over completely.

We also learned that SPDY is served over SSL. Always. Finally, we found that SPDY is definitely *not* meant for simple `hello world`. To find out what kind of applications SPDY is good for, keep reading. Things are going to get a lot more interesting.

# Chapter 3. SPDY and the Real World

SPDY ain't optimized for `hello world`.

SPDY is meant for the real world. The world of inexplicable network delays and dropped connections. A world in which applications are getting more complex and demand for quickly loading sites is ever increasing.

It is nice knowing that SPDY can be introduced with relative ease to existing sites. Or at the very least, that getting started with SPDY apps does not require much new understanding beyond what we already know.

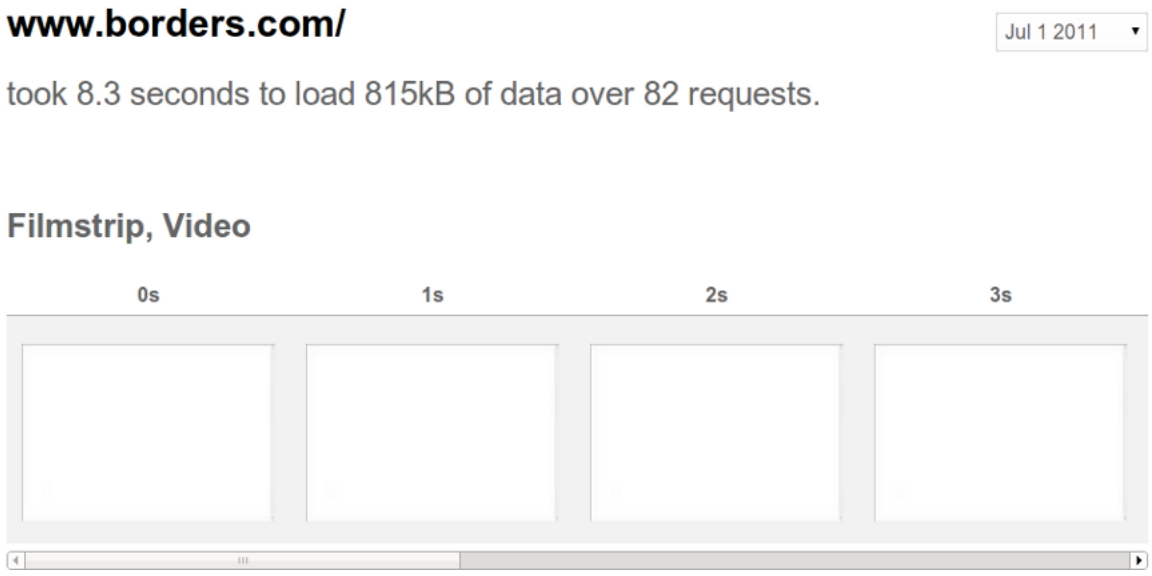
The burning question is, how does SPDY improve life in the real world?

## 3.1. A Tale of Two Websites

The most important click on any site is the first one. The old adage that you never get a second chance to make a first impression is crazy important on the web. If you greet your customers with a page that takes 5 seconds to respond, or if your super new HTML 5 + backbone.js + node.js app forces excited users to stare at a blank page while everything gets initialized, then you are begging customers and potential customers to look elsewhere.

Consider, if you will, a tale of two websites. On July 1, 2011, if you looked at the Borders.com website for 1, 2, 3, 4 seconds, this is what you would have seen [1]:

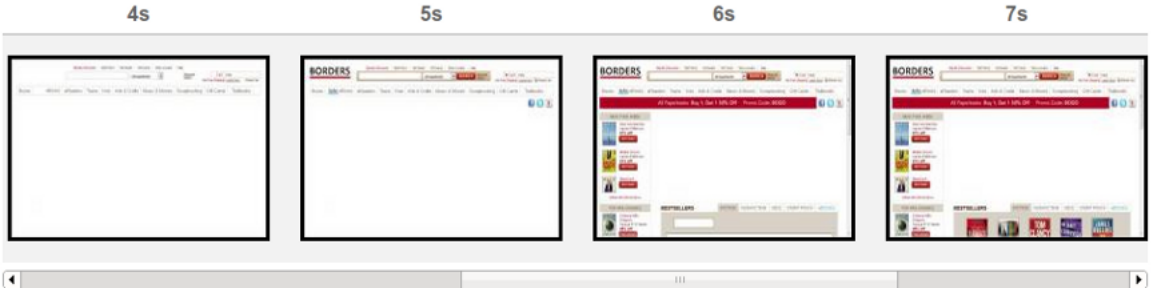
[1] from the very cool HTTP archive site: http://httparchive.org/viewsite.php?u=http%3A%2F%2Fwww.borders.com%2F&l=Jul%201%202011




Looks like I forgot a graphic doesn't it?

That's because Borders.com responded to potential customers with *nothing*. For four seconds, customers saw nothing.

In the next four seconds a potential Borders.com customer saw:

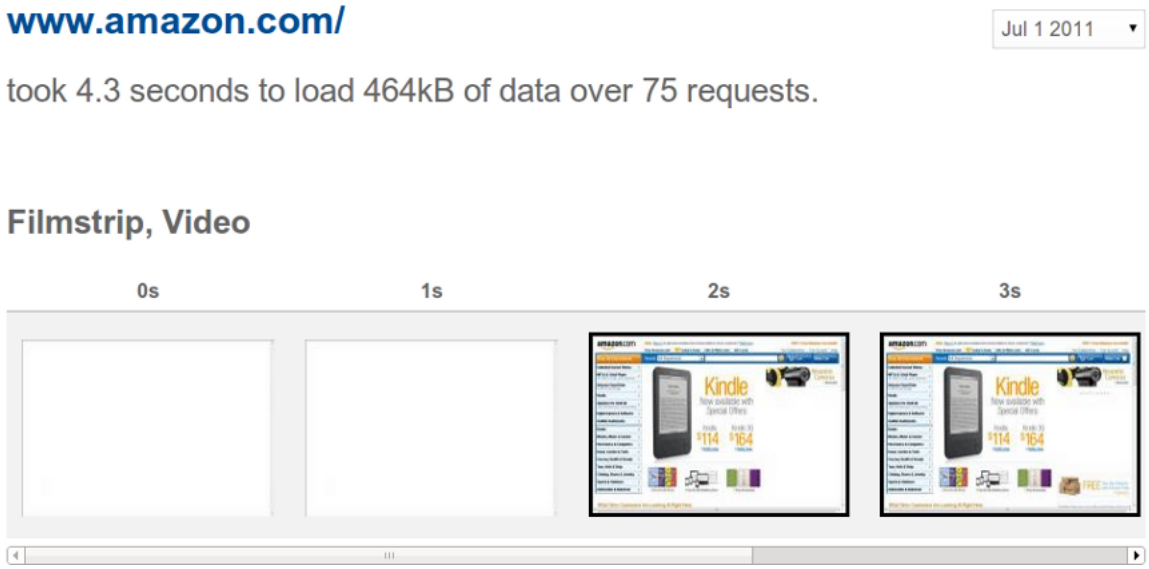



The top-level navigation finally shows after 4 seconds. By the end of the seventh second, the main content has *still* not made an appearance. Considering that a third of broadband users will abandon a site after 4 seconds, there is little wonder that Borders is out of business.

*Borders simply didn't care about their customers.*

How can you argue otherwise? In **2011** Borders failed to take even rudimentary steps towards improving the experience of their online presence. This was not back at the turn of the century when we were still learning how the web worked. It was 2011 and Borders did nothing to earn customers. They deserved bankruptcy.

Borders' history will forever be entwined with Amazon.com. And by "forever", I mean until we have all forgotten Borders completely next year. But consider what a potential Amazon.com customer sees when accessing the website [2]:




After 2 seconds, the *entire* page is done rendering. How does Amazon do it? They work *really* hard to overcome the limitations of HTTP. They do everything from "minifying" and combining Javascript files to parallelizing requests across multiple domains. They painstakingly verify that *all* static resources are cacheable. They parallelize requests across multiple hostnames. They even ensure that their static content is served without cookies.

Amazon works really hard to give existing and potential customers the best possible experience. As web developers and administrators, shouldn't we all?

[2] http://httparchive.org/viewsite.php?u=http%3A%2F%2Fwww.amazon.com%2F&l=Jul%201%202011

But what if we could do it all, without all the hard work? This, of course, is SPDY's promise.

# 3.2. A Little Less Hello World

To see this in action, let's revisit our `hello world` application. First a vanilla HTTP version of a `real world` app:

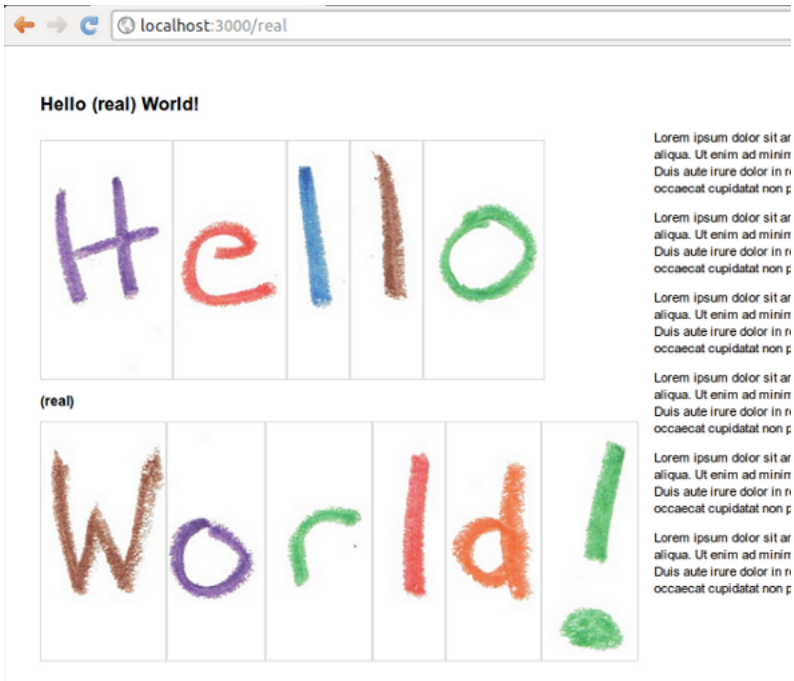
```
app.get('/real', function(req, res){
  res.render('real', {
    title: 'Hello (real) World!'
  });
});
```

More important than the real world express.js routing, is the content of the web page itself. In the `hello world` page, there was nothing but HTML and text (and very little of either). On the `real world` page, let's add references to a dozen or so images, some non-trivial CSS, a minified version of the jQuery library and ~3kb of dummy text:

```
<html>
<head>
<title>Hello (real) World!</title>
<link rel="stylesheet" href="/stylesheets/style.css"/>
<script src="/javascripts/jquery-1.6.2.min.js"></script>
</head>
<body>
  <h1>Hello (real) World!</h1>
  <div id="hello">
  <p>
    <img src="/images/00-h.jpg"/>
    <img src="/images/01-e.jpg"/>
    <img src="/images/02-l.jpg"/>
    <img src="/images/03-l.jpg"/>
    <img src="/images/04-o.jpg"/>
  </p>
  <h2>(real)</h2>
  <p>
    <img src="/images/06-w.jpg"/>
    <img src="/images/07-o.jpg"/>
    <img src="/images/08-r.jpg"/>
```

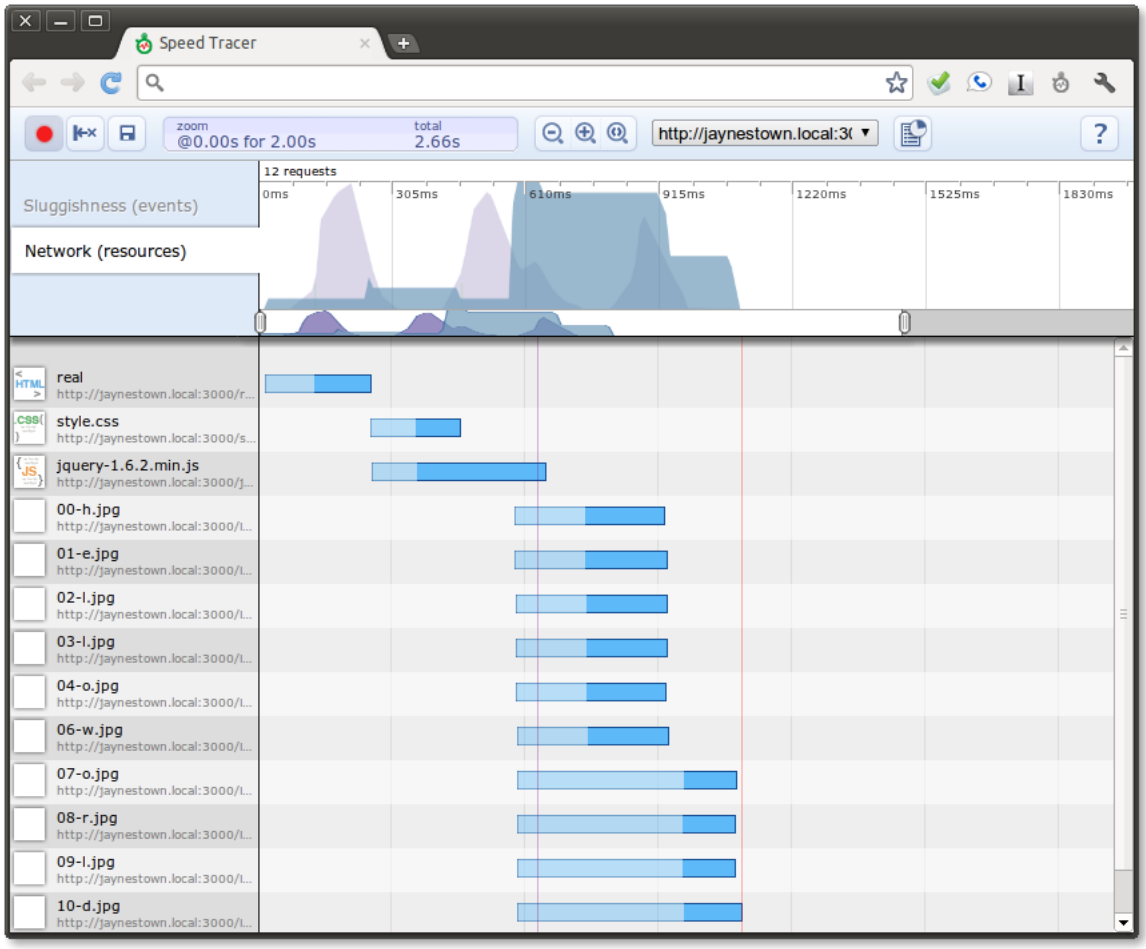
```
  <img src="/images/09-l.jpg"/>
  <img src="/images/10-d.jpg"/>
  <img src="/images/11-bang.jpg"/>
</p>
</div>

<p>Lorem ipsum dolor
…
```

The assembled page may not win awards, but it is a slightly more accurate representation of the structure of a typical web page:



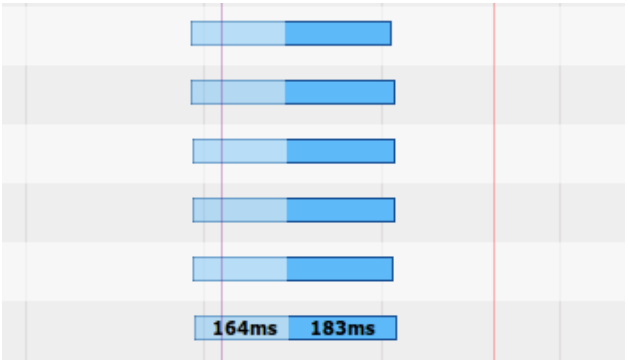

This is where things start to get interesting.

Looking at network requests in the Speed Tracer Chrome extension, we find:



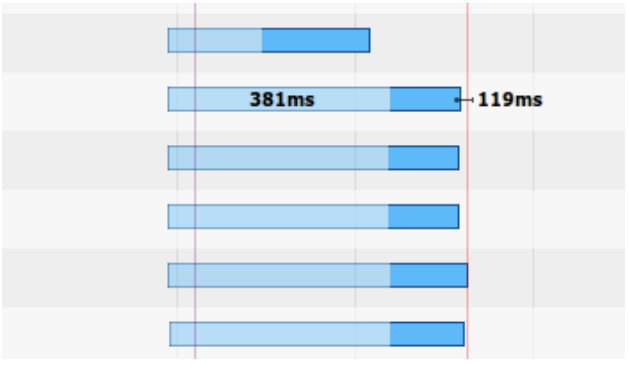

Just as Chrome starts to parse the request for the `/real` resource, it sends out two requests for each item in the `<head>` section of the HTML document (the CSS and Javascript).

After sending and getting back the responses, the browser then requests *six* additional resources—the first six images in our real world HTML. A closer examination of those six requests reveals that they are all quite similar in request and response time:




Somewhere around 150ms after the request for these images, the server response finally starts to reach the browser. 180ms or so later, the images have arrived in their entirety.

The remaining images are loaded concurrently as well:



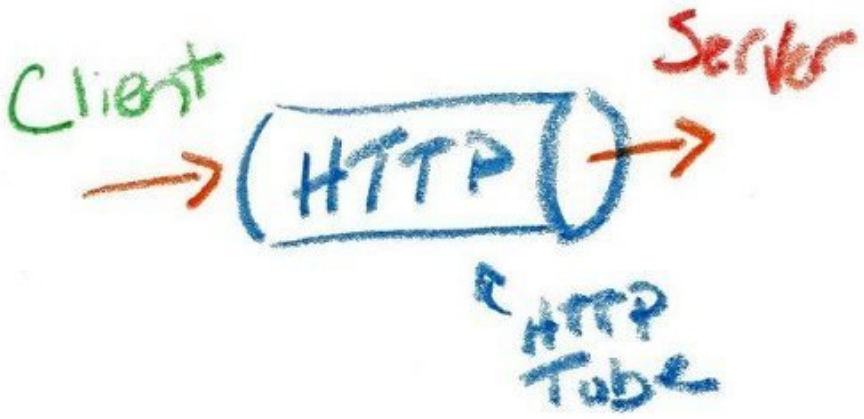

The browser has to wait 380ms before these images start returning. Interestingly, these images do not take quite as long to transfer as the previous six. Whereas the first six images took around 180ms, these images are on the order of 120ms—a savings of 60ms.

We are still looking at plain, old HTTP/1.1 here. None of this is unique to node.js. You will find similar numbers with PHP, DotNet or any other web server in existence. Which begs a few questions. Why are only six of the images loaded at a time? Why not all of them? Why does the second group transfer 33% faster? And how are all of these resources able to transfer at the same time since HTTP is a serial protocol?

The answer to all of those questions is quite simple, really…

# 3.3. The Internet is Made Up of a Series of Tubes

If the internet is built out of tubes (bear with me on this one), then a connection between client and server might look like:

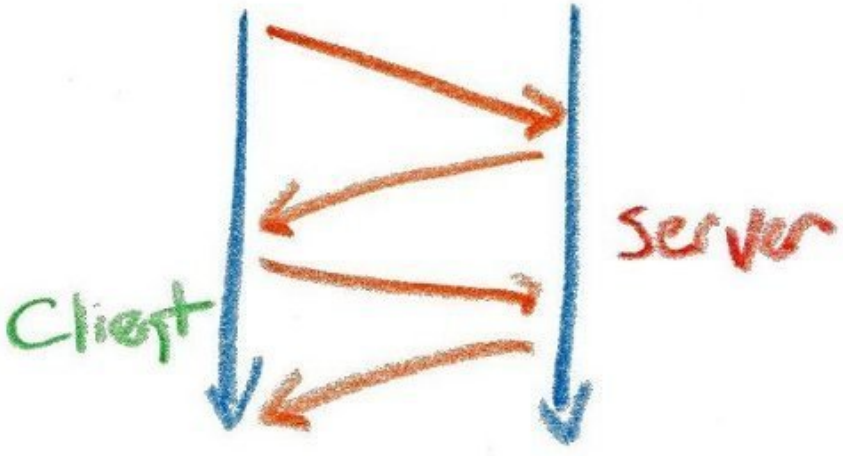



## 3.3.1. Browsers Have 6 Tubes

Even as far back as the 1900s, web browsers have opened more than one connection to a web server. In fact, all modern web browsers open 6 connections. Why?

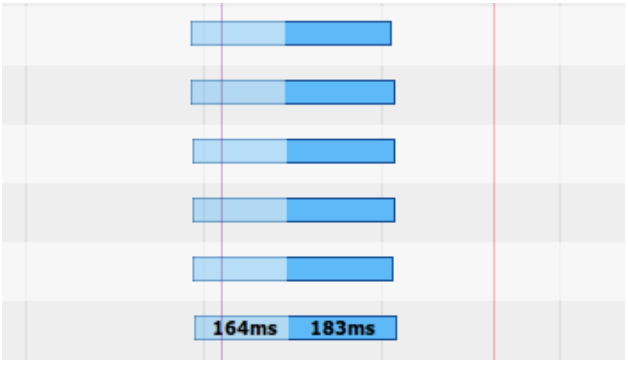
The 6 connections get around a fundamental limitation of HTTP: requests for resources block subsequent requests. That is, if my browser is requesting the homepage on a connection, it cannot use that same connection to request a background image or a stylesheet until the web server completes sending back the homepage.

HTTP interweb tubes block until a response comes back.




Browser vendors saw the inefficiency of this and so built in 6 connections. As soon as the first bits of a page are returned, browsers begin looking through those bits for stylesheets and other associated resources. As soon as they see such a beast, the browser makes a request for them—on a separate connection.

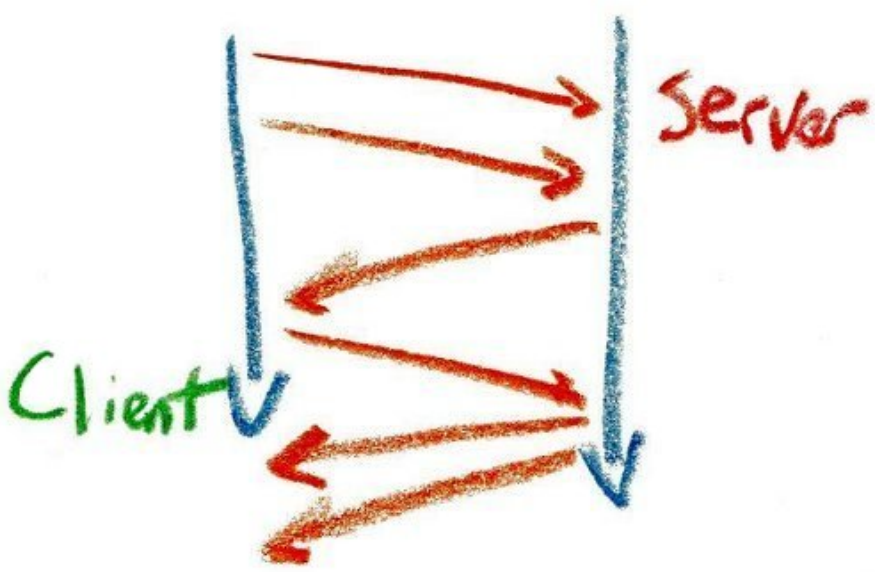
This then, explains why the browser requested only the first six images on the `real world` page:




Six requests are all that a browser is capable of making at the same time. And, since each of those tubes is an HTTP tube, they have to block until the web server response has completely reached the browser.

## Note

HTTP/1.1 included the concept of pipelining (section 8.1.2.2 of RFC 2616 [3]). Pipelining allows the browser to make a request on the same tube *before* it has received the complete response to a previous request.



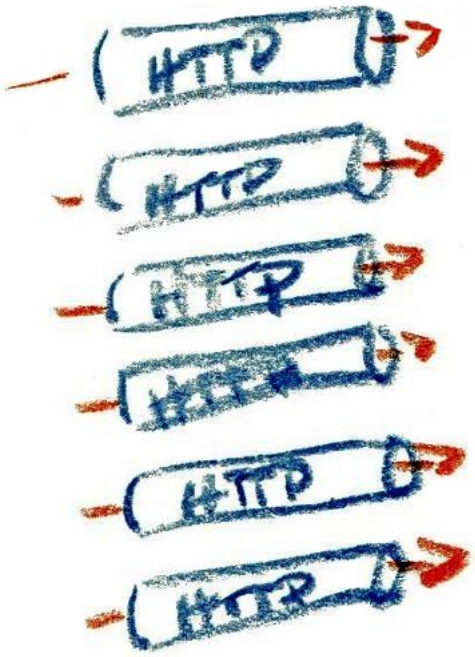

The main drawback to pipelining is that no browsers support it. Well, Opera is reported to support it in weird edge cases, but we may as well agree that no web browsers support it.

It is a testament to the authors of HTTP/1.1 that, in addition to all of the other great stuff in there, they included pipelining. And more than a little disappointing that their foresight was rewarded so dismally.

We will talk more about pipelining in Chapter 9, *SPDY Alternatives*.

In effect, modern browsers have a series of tubes (turns out that guy wasn't so crazy after all) that look something like:

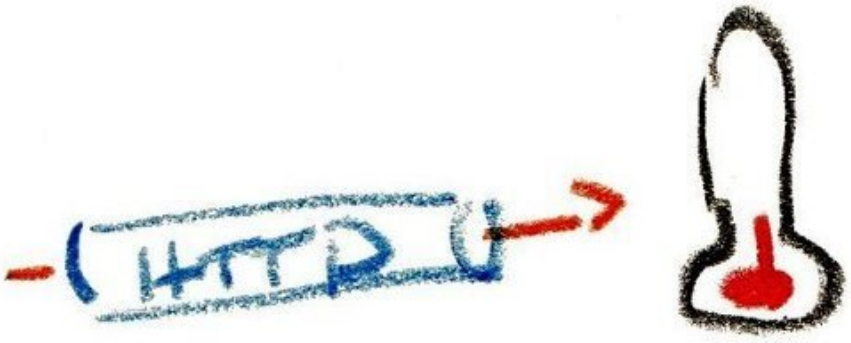
[3] http://www.ietf.org/rfc/rfc2616.txt




## 3.3.2. HTTP Tubes are Cranky Old People on a Cold Morning

These interweb tubes are petulant little things. They do not like being asked to do stuff. Once they are in action, they can positively jam data through themselves. But when they first open, they are creaky old bones on a cold winter morning. They are cold interweb tubes.

Watch out for cold tubes.

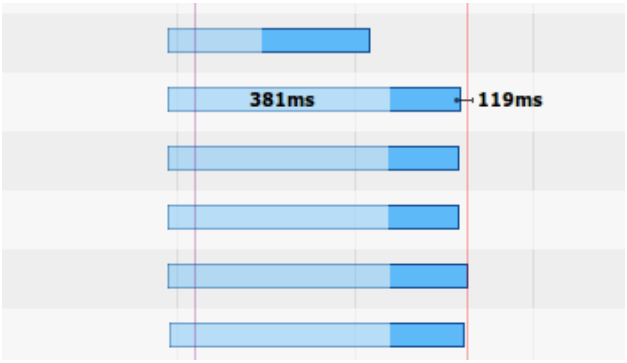



HTTP is built on top of TCP/IP. The latter was designed to work with asynchronous data conversations that could potentially handle large amounts of data. "Potentially" is an important word because when TCP/IP connections are first initiated, they have a very small congestion window (CWND). The congestion window describes how much data either end of the TCP/IP conversation will send at a given time.

TCP/IP is built so that, over time, the CWND grows to fill the available bandwidth between client and server. The growth rate is referred to as "slow start". It is actually quite fast (can be on the order of tenths of a second depending on the round trip time). The name "slow start" is historical (it used to be even slower).

But in the case of HTTP, "slow start" is quite appropriate because it has a significant impact on how quickly a given resource can be downloaded. A moderately complex web page might compress to 2kb in size. Over a 3Mbit/s connection, that ought to be downloaded in tens of milliseconds. In practice, however, the CWND needs to warm up before the full bandwidth is used, which might push the download time to something closer to a second. Factoring in packet loss (which itself negatively impacts CWND) and latency between client and server and we start to see actual page load times.

TCP/IP slow start explains why the second set of images took 33% less time to transfer than the first set of images:




By the time the browser got around to requesting those images, the interweb tubes that it was using had warmed up. They may not have warmed up completely by this point, but they had warmed to a point at which they can *significantly* improve transfer times.


### Tip

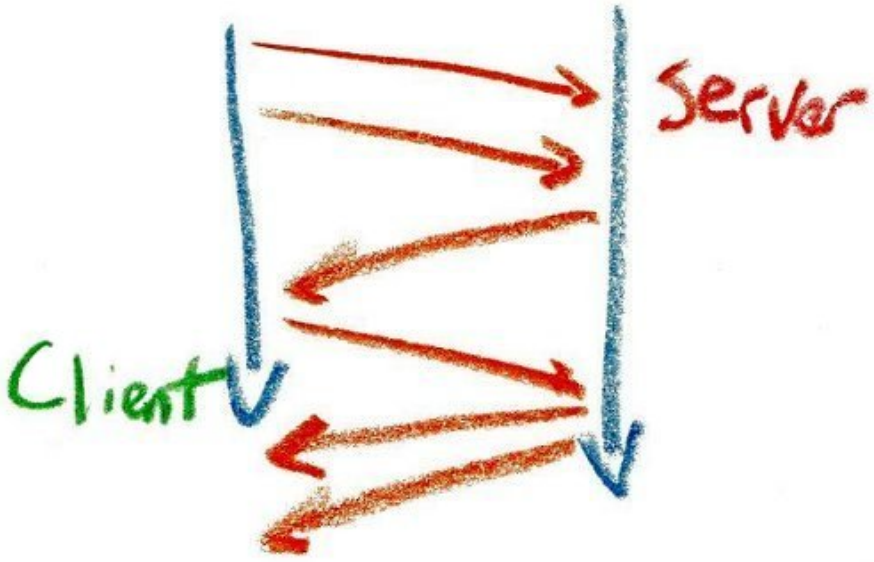
*Lesson #4:* Warm interweb tubes are happy interweb tubes.

But now we have ourselves a catch-22. And it is this catch-22 that proves to be the crux of why simply increasing bandwidth will *not* decrease page load times. We need multiple tubes to be able to request all of the resources needed to quickly render pages. BUT…

But each of those tubes needs to be warmed. As each tube is warmed, it is contending with the other tubes, along with other network resources, for the full bandwidth for a user. Any errors or dropped packets may freeze out a tube for a time resulting in less-than-ideal user experience.

As pages become more complex, we are only going to demand more of these tubes. This demand risks further errors and limits the ability of browsers to compensate.

This is why SPDY does away with the 6 tubes completely. Instead, it goes back to a single pipelined tube. It acknowledges the foresight of the original HTTP/1.1 authors and allows client and server to exchange requests whenever convenient.




By going to a single pipelined tube, SPDY has the benefit of almost immediately warming a tube without the worry of warming 5 others. The result is that a SPDY tube warms faster and has less contention with other network resources.

### 3.3.3. Why Not Start the Tubes Warm?

OK fine. You might be willing to stipulate that the internet really is made of tubes. You might even go along with the idea that tubes need to be warmed up a little bit before they can work at full capacity.

But if all of that is true, why not just start the tubes warm? Surely browsers and servers can muck with the TCP/IP packets to increase the initial CWND. In fact they can and do [4].

As we will see in Chapter 9, *SPDY Alternatives*, newer versions of Firefox are experimenting with CWND on the client side.

SPDY argues that this is a bad thing and the argument is quite persuasive.

Recall that all modern browsers already open 6 connections to a host. Put another way, browsers have *always* increased CWND. If the initial CWND would otherwise be 1000 bytes, the initial CWND is effectively 6000 bytes for browsers.

In effect, we are already subverting the intention of TCP/IP congestion control schemes on which the Internet has been built. Increasing the initial CWND on top of this is only going to exacerbate the abuse of TCP/IP.

We already experience packet loss daily and the occasional site that just stalls. A reload usually resolves things, but are things going to get better if we increase TCP/IP abuse?

Even if they do, we still have none of the other capabilities of SPDY.

## 3.4. SPDY and the First Click

All this talk of interweb tubes and CWND is all very good, but what does SPDY actually do for a website?

To answer that, we take our "real world" sample application:

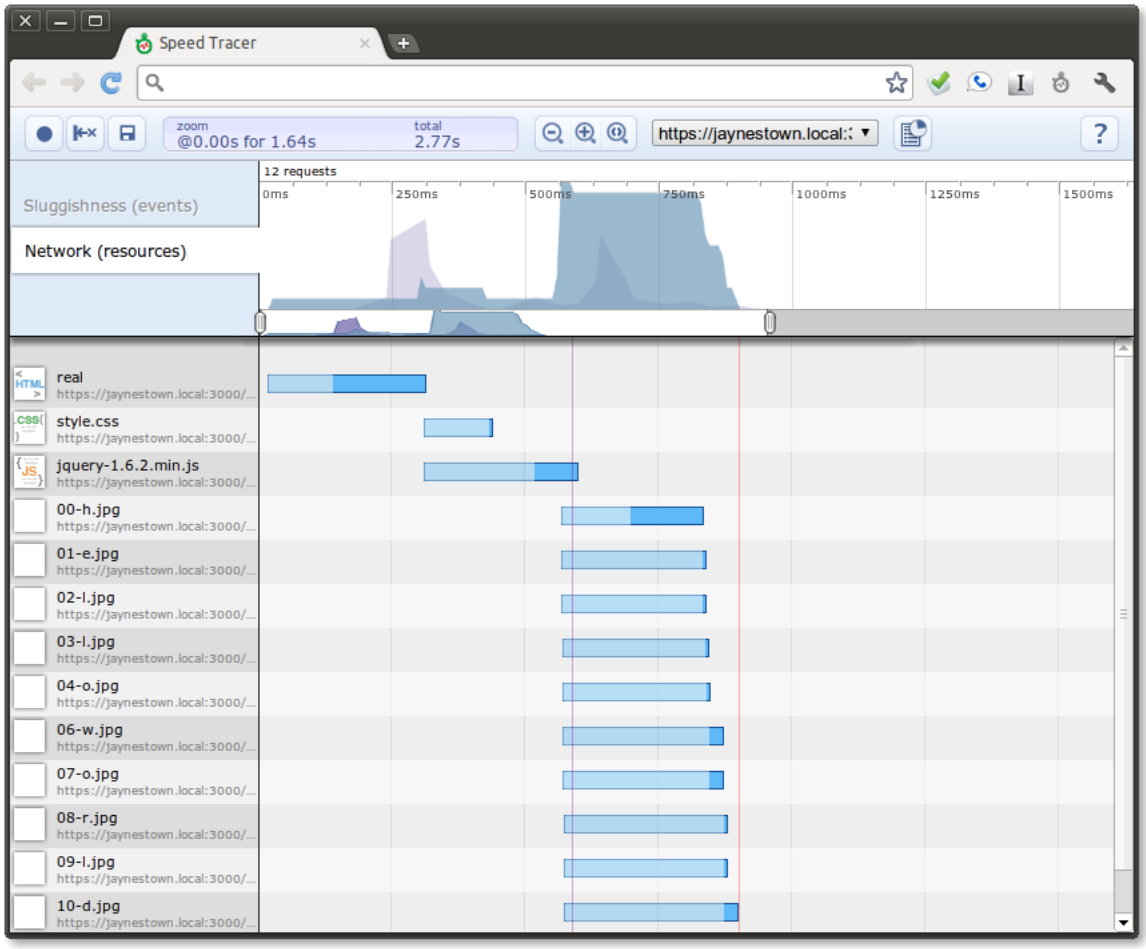
[4]There is even a draft proposal to increase initial CWND across the board

```
app.get('/real', function(req, res){
  res.render('real', {
    title: 'Hello (real) World!'
  });
});
```

And we add SPDY configuration to it:

```
var express = require('express-spdy')
  , fs = require('fs');

var app = module.exports = express.createServer({
  key: fs.readFileSync(__dirname + '/keys/spdy-key.pem'),
  cert: fs.readFileSync(__dirname + '/keys/spdy-cert.pem'),
  ca: fs.readFileSync(__dirname + '/keys/spdy-csr.pem'),
  NPNProtocols: ['spdy/2', 'http/1.1']
});
```

Accessing the page now looks like:



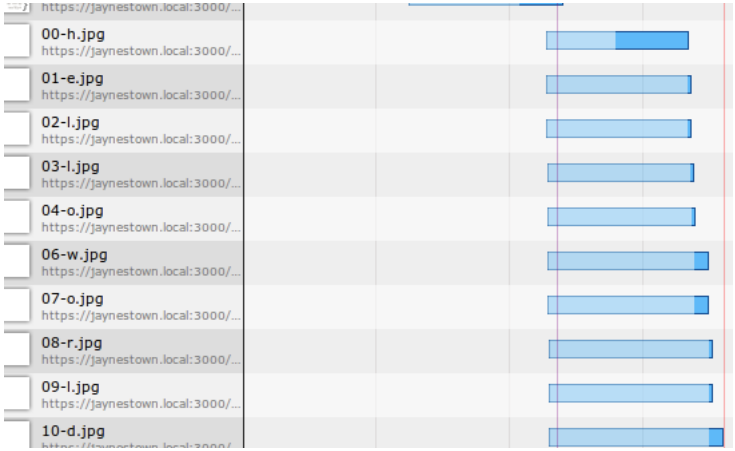

Comparing that with the vanilla HTTP network session to access the same page, a few things stand out. First, the request for the web page itself takes a little longer than over HTTP. This is to be expected since SSL negotiation is involved.

SPDY does not gain much on HTTP with the next two requests. Loading in the `<head>` resources is mostly occupied by waiting on the network. The round trip time affects SPDY just as much as it does HTTP. Even so, SPDY does begin to make up some ground by virtue of a warm tube:


The transfer of jQuery took a little over 110ms via a cold HTTP tube. Since the SPDY tube is warming, it only takes 80ms.

And, after another relatively long transfer for the first image, the remaining packets transfer over a warm SPDY tube. When they transferred over HTTP, each image still took around 100ms to get back to the browser.

With warm SPDY tubes, the transfer times are generally less than 10ms:




The total transfer time for the HTTP page was 1300ms. Over SPDY, the same page transferred in 900ms. That is a 33% decrease in page load time—even with the SSL penalty.

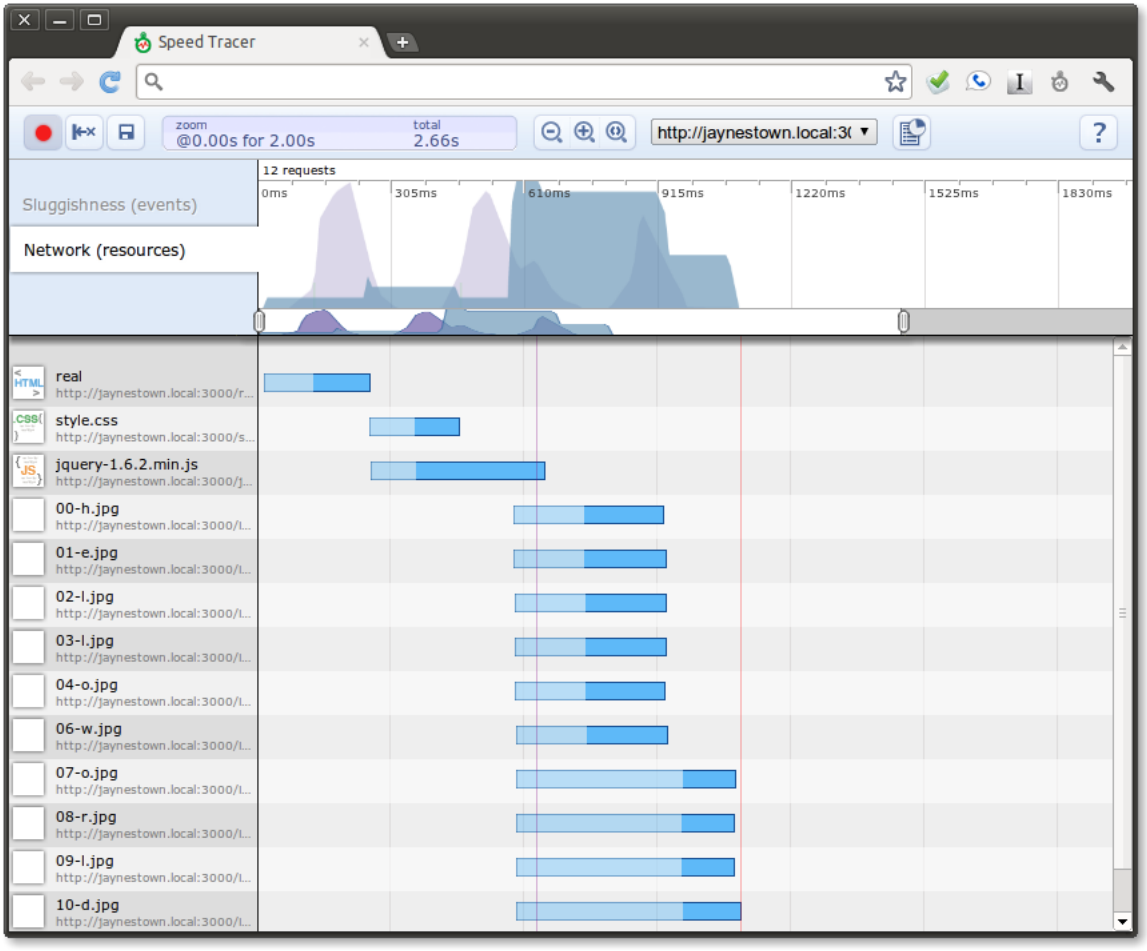
That is not too bad for simply adding a few lines of code. But of course, we can do even better. Let’s have a look at that in the next chapter.

# Chapter 4. SPDY Push

So far we have taken our relatively simple "real world" application and achieved a very satisfying decrease in page load time. There has been relatively little new code to add to the application. For the most part, we are doing little more than adding SPDY configuration to a vanilla HTTP application.

That is by no means the end of the SPDY story.

Recall from the previous chapter that the network traffic for an HTTP page looks like this:



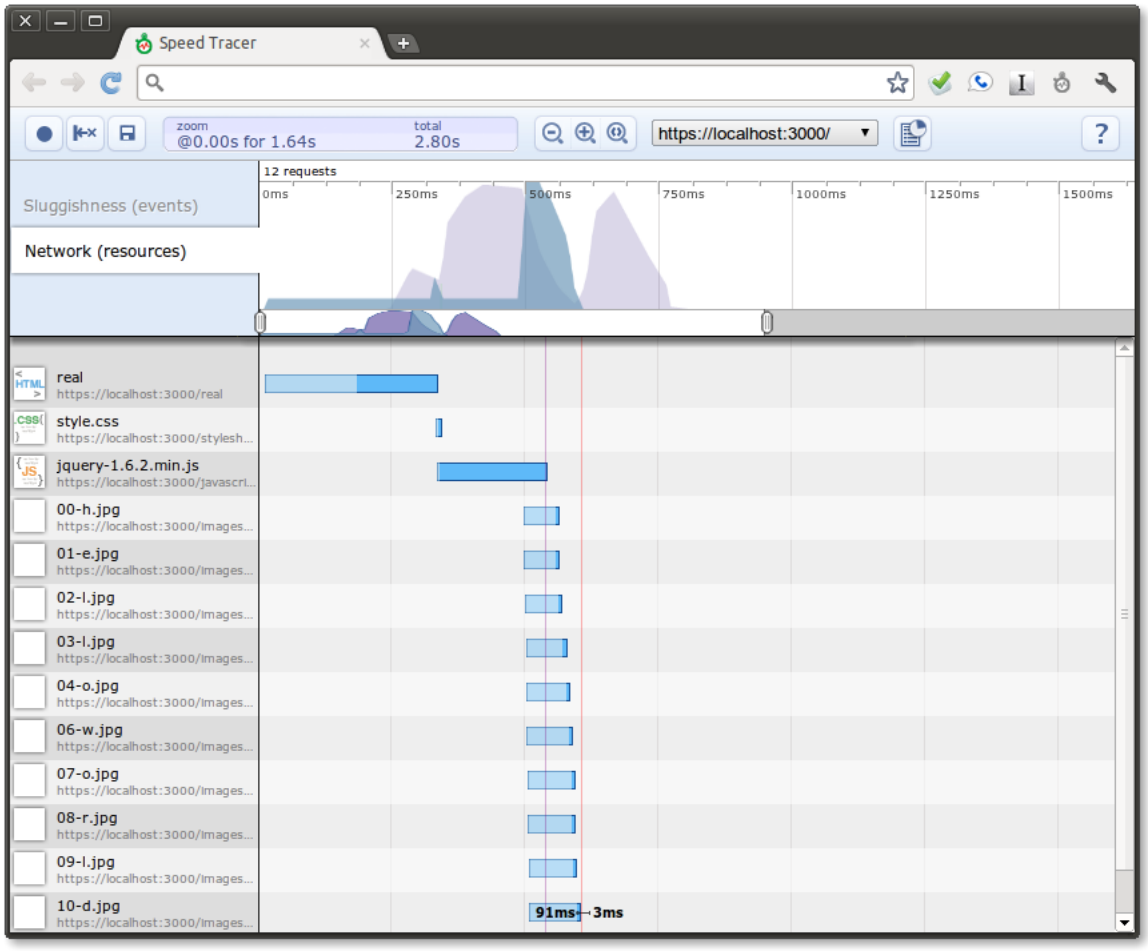
The request/response bars in the real world network diagram are more or less equal on both sides. That is, as much time is spent waiting for a response to the request as is spent waiting on the response to complete.

By enabling SPDY, we shrank the dark blue side of the request/response bars to tiny little things. SPDY does this by slamming all of its data on a single HTTP interweb tube. A single interweb tube overcomes TCP/IP slow start quicker than a bunch of less used tubes that compete with each other for network resources.

A 33% decrease is nothing to sneeze at, but what else can SPDY possibly do? The light blue bars, the time waiting on a response, cannot get any smaller. Network latency is network latency. SPDY cannot magically make a delay of 50ms become 5 ms. Right?

Actually that is correct. SPDY can’t fix your network. But it can get around it.

Before looking at how this is done, let’s look at the results:




That is stunning.

The request for the `/real` resource still takes a while—about 100ms longer than vanilla HTTP due in part to SSL overhead. But for every other request there is *no* overhead for requests.

Zero.

### Note

Speed Tracer is not always reporting when the request was sent. Rather, the light blue timer is started when the request hits Chrome's internal network queue. It may be a while before the request actually hits the network. So just bear in mind that the delay is not always network related—it may be internal to the browser as well.

So how does SPDY achieve this? I just admitted that SPDY cannot overcome network limitations but now we see that it magically made the request overhead disappear.

The answer to this mystery is one of SPDY's coolest, but easily misunderstood, features.

# 4.1. SPDY Server Push

When web developers think of server push, we usually think of things like Comet, AJAX long-polling, web sockets or similar concepts.

This is *not* what SPDY server push is.

You could argue that the SPDY spec misnamed server push, but let's find out what it is first and then decide.

SPDY is an asynchronous protocol, which means that client and server can both initiate new streams. A browser needs to initiate a new stream for every new request that it makes. It reuses the same interweb tube for the duration of the session, but each new request is a new stream within that session.

So when might a server need to initiate a stream? SPDY servers can initiate new streams when they want to push data directly into the browser cache.

Let me repeat that because this is completely new: SPDY servers can push data directly into the browser cache.

### Important

SPDY's sole purpose is to decrease page load times. Replacing Comet, AJAX long polling and web sockets is **well** outside of that mission statement. Certainly, the

specification could eventually be updated to support this kind of push. For now, the authors of the spec are being entirely pragmatic by sticking to a single guiding principle.

Web developers have been creating ever more complex applications to meet the demands of the discerning customer. As these applications have increased in complexity, the web servers that serve up static content (Javascript, CSS, images, etc.) have remained back in a 1994 mindset. If, **and only if**, a browser requests a resource, does the server send the resource back.

Pity the poor web server. It is usually a honking machine. It is perfectly capable of recognizing that, every time somebody requests the homepage, they always ask for thirty images, various Javascript files and some stylesheets. And yet a machine spec'd out to the maximum that money can buy is still relegated to a dumb server.

# 4.2. SPDY is the Smart Server's Best Friend

Browsers do not care how data got in their caches.

That is an odd concept to wrap your brain around the first time, because, until now, there has only been one way to get data in the browser cache.

But SPDY is able to jam lots of data over that warm SPDY tube—directly into the browser cache. Whenever a browser goes to request a resource, it first checks its local cache. If the resource is already in cache (and hasn't expired), then the browser simply uses that resource instead of incurring the round trip time needed to obtain the desired object from the server.

So the server, tired of being a dumb file server, is finally liberated to use the brains granted it. If the browser requests the homepage, the clever server can now send along all of the associated resources in addition to the homepage.

The result is a complete lack of round trip request time and *substantially* decreased page load time. In the case of our simple web page served across a relatively slow network

(100ms round trip time), we see a decrease in page load time of more than 50% as the request time drops from almost 1300ms to 600ms!

There are some considerations with SPDY server push but, before we get to those, let's have a look at the code behind SPDY server push. Surprisingly, there are no changes to the routes that describe the various resources. The "real world" resource remains no more than:

```
app.get('/real', function(req, res){
  res.render('real', {
    title: 'Hello (real) World!'
  });
});
```

To get SPDY server push working, additional configuration is required. We tell the SPDY server how to push directly in the `createServer` call with a new `push` callback function:

```
var app = module.exports = express.createServer({
  key: fs.readFileSync(__dirname + '/keys/spdy-key.pem'),
  cert: fs.readFileSync(__dirname + '/keys/spdy-cert.pem'),
  ca: fs.readFileSync(__dirname + '/keys/spdy-csr.pem'),
  NPNProtocols: ['spdy/2'],
  push: simplePush
});
```

The `simplePush` function tells the server when and what to push:

```
function simplePush(pusher) {
  // Only push in response to the first request
  if (pusher.streamID > 1) return;

  pusher.push([
    local_path_and_url("stylesheets/style.css"),
    local_path_and_url("javascripts/jquery-1.6.2.min.js"),
    local_path_and_url("images/00-h.jpg"),
    local_path_and_url("images/01-e.jpg"),
    local_path_and_url("images/02-l.jpg"),
    local_path_and_url("images/03-l.jpg"),
    local_path_and_url("images/04-o.jpg"),
    local_path_and_url("images/05-space.jpg"),
    local_path_and_url("images/06-w.jpg"),
    local_path_and_url("images/07-o.jpg"),
```

```
    local_path_and_url("images/08-r.jpg"),
    local_path_and_url("images/09-l.jpg"),
    local_path_and_url("images/10-d.jpg"),
    local_path_and_url("images/11-bang.jpg")
  ]);
}
```

The "simple" in `simplePush` refers to the first two lines that limit pushing to the first request:

```
  // Only push in response to the first request
  if (pusher.streamID > 1) return;
```

The `pusher` object exposes the request URL and other attributes that let the server be a bit more intelligent about pushing. For our purposes, pushing on the first request is a good enough solution.

After that, it is a simple matter of telling the server where to find the resources locally and with what URL the browser should store the resource.

In total, we have added around 30 lines of code to our real world express.js application. For those 30 lines of code, we have decreased the page load time from 1300ms to 600ms —an amazing 55% decrease in page load time.

That is a huge decrease in page load time that is **easily** noticed by any potential customer. Moreover, a single pipe is more robust than 6 pipes competing for network resources.

### Tip

*Lesson #5:* Always favor a warm pipe and cheat as much as possible. Your customers won’t complain.

The code that instructs `express-spdy` what to push is little more than a list of resources:

```
  pusher.push([
    local_path_and_url("stylesheets/style.css"),
    local_path_and_url("javascripts/jquery-1.6.2.min.js"),
    local_path_and_url("images/00-h.jpg"),
    // ...
    local_path_and_url("images/11-bang.jpg")
```

```
  ]);
}
```

The function `local_path_and_url` is specific to this express-spdy app and is not part of express-spdy proper. It tells express-spdy where to find the resource on the file system and what URL to use in the browser's cache:

```
function local_path_and_url(relative_path) {
  return [
    "public/" + relative_path,
    "https://jaynestown.local:3000/" + relative_path
  ];
}
```

As we will see in later chapters, there are actually more tricks that you can play to get even more performance out of SPDY server push. The results in those cases will not be quite so dramatic, though the implications on what is possible may prove to be even more startling.

To understand how that is possible, however, first requires a look at some of the details of the SPDY protocol itself. Once we have done that, we will pick back up with our express-spdy application.

For now, we have taken a relatively nondescript web application—that is by no means poorly written—and decreased the response time by well over 50%. For all of that change, there has been relatively little required at the application layer. By jamming everything over a single, asynchronous pipe, we overcome TCP/IP's objections to HTTP and do so very quickly.

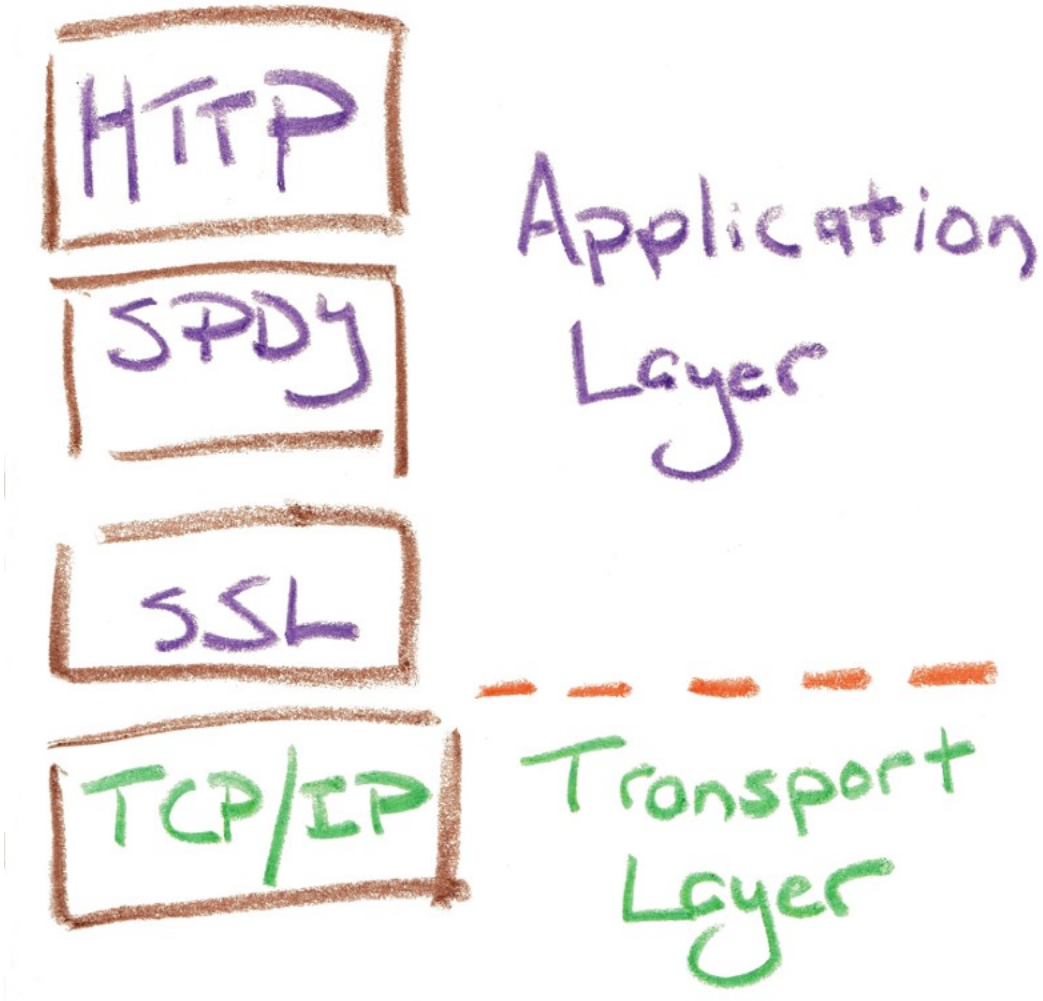
By pushing resources that we know need to go along with a requested page, we not only jam that data at a high speed, but we do so without the overhead of dozens of round-trip requests. All of this with the promise of a more robust connection to the web server.

There is almost no downside to SPDY support. There are some fundamental differences between it and HTTP. To understand how those differences allow web developers to make applications so fast, we first need to peek under the covers at actual SPDY data, which is what we do next…

# Chapter 5. The Basics

At this point we have a strong feel for what SPDY is like in applications. For all of its power, it is surprisingly unchanged from normal HTTP application development. Now is a good time to take a closer look at the protocol itself to see how it makes this possible.

SPDY sits in between HTTP and SSL in the network stack:



Back in Chapter 2, *Your First SPDY App*, we were introduced to SPDY's prime directive: make page load times faster. There is a corollary to this prime directive. A secondary directive, if you will.

**Tip**

*Lesson #6:* SPDY wants to be adopted everywhere. It reuses existing network infrastructure so that no fundamental networking changes are required.

Most of your HTTP knowledge will serve you well as you explore SPDY. If you do not worship at the altar of RFC 2616 [1], don't worry. I'll keep it at a level that it should make sense.

# 5.1. Yes, It is a Binary Protocol (Deal with It)

Unlike HTTP, SPDY is a binary protocol. This is, admittedly, a loss.

A huge factor in the success of HTTP was its plain text nature.

Can't figure out why a web page is behaving that way? A quick `tcpdump` will show you the request and response headers to debug. You can even grab the body of the response as it comes through.

Need to troubleshoot a particular resource? Telnet into the web server and you can simulate a web session by hand. I still get a kick out of doing that (admittedly, it's even more fun with SMTP).

But guess what? The plain text web is dead.

It grew ill when a dozen layers of complexity were added to your site to enforce authentication and interoperability. It is impossible to access your site without cookies and MD5 signatures—not the kind of thing easily done over telnet.

[1] RFC 2616 is the HTTP specification, permanently located at http://www.ietf.org/rfc/rfc2616.txt

**Firesheep**

Firesheep (codebutler.com/firesheep) is a plugin for the Firefox web browser. It sniffs unencrypted packets on the local network for cookies that hold currently active session IDs. Using those session cookies, Firesheep allows the user to take over anyone's active session—doing anything that the legitimate user can do.

Hopefully that sounds scary.

The plain text officially died when Firesheep hit the interwebs and laid bare that stealing web sessions was not only doable, but *trivial*.

This is the age of SSL for even the most basic sites. This is the age of GZIP compression and complex authentication. The plain text web was a wonderful thing, but it is a thing of the past.

Binary is the future. And SPDY is a binary protocol.

If we embrace binary, there is much to be gained. A significant amount of overhead in HTTP comes from redundant headers. Compressing these headers into a tightly packed binary format is a big win for SPDY.

SPDY is a highly structured protocol. The frames will almost always look alike—regardless of the type of frame or the state. In practice, this is a great benefit as you get familiar with packets.

# 5.1.1. Data Frames

Data transmission in SPDY is so trivial that it is almost laughable. But within that simplicity is quite a bit of power. It begins with 4 octets of data that describe the stream ID to which it belongs.

**Note**

I adopt the node.js convention of referring to 8bits as an octet rather than byte. The term "byte" has become so overloaded in normal usages that octet is preferred for its unambiguity.

Regardless of the term, the maximum number represented by an octet is `1111 1111`. In decimal, this is `255`. In hexadecimal, this is `0xFF`. In this book, nearly all discussions of binary data will use hexadecimal.

After that comes 4 more octets (1 for flags and 3 for the length of the data being sent) and then…

Data.

Raw binary data is packed into frames and sent over the wire. The data might be raw HTML or PNG images. In early drafts of the SPDY spec, the data could be compressed by SPDY itself, but, more likely the server will use GZIP compression to send data.

### Note

The switch from SPDY compressing the data to leaving this in the application layer was an acknowledgement by the SPDY authors that SPDY does just enough to accomplish what it needs to and no more. The application layer has been performing GZIP compression for years, so why re-implement at the transport layer?

A data frame might look something like:

```
000000020000022b626f6479207b0a202070616464696e673a20303b0a20
206d617267696e3a20303b0a7d0a0a73656374696f6e236d61696e207b0a
20206d617267696e3a2030206175746f3b0a20206d61782d77696474683a
2036303070783b0a2020746578742d616c69676e3a2063656e7465723b0a
20206261636b67726f756e643a2075726c28737064792e6a706729206e6f
2d726570656174203530252035302530253b0a2020686569676874743a20333030
70783b0a7d0a0a73656374696f6e236d61696e2068312e7469746c65207b
0a2020636f6c6f723a20233333333b0a7d0a0a73656374696f6e236d6169
6e20646976236e6f7465207b0a2020666f6e742d73697a653a2031357078
3b0a2020746578742d736861646f773a203020302030347078207267626128
3235352c3235352c3235352c302e35293b0a20206261636b67726f756e64
3a207267626128302c302c302c302e36293b0a2020646973706c61793a20
696e6c696e652d626c6f636b3b0a2020636f6c6f723a2077686974653b0a
202070616464696e673a203130707820357078 3b0a20206d617267696e2d
626f74746f6d3a20313770783b0a7d0a0a73656374696f6e236d61696e20
64697623636f6e74656e74207b0a2020746578742d736861646f773a2030
203020357078207267626128302c302c302c302e33293b0a2020666f6e74
```

```
2d73697a653a20323570783b0a20206261636b67726f756e643a20726762
61283235352c3235352c3235352c302e36293b0a7d0a0a
```

Yikes! That is a lot of numbers and letters that seemingly mean absolutely nothing!

It is easier to understand any SPDY frame by deconstructing it into separate octets and matching it against the SPDY spec ASCII representation of data frames:

```
             +----------------------------------+
00 00 00 02  |C|       Stream-ID (31bits)       |
             +----------------------------------+
00 00 02 2b  | Flags (8)  |  Length (24 bits)   |
             +----------------------------------+
62 6f 64 79  |               Data               |
20 7b 0a 20  +----------------------------------+
20 70 61 64
....
```

In this case, we are looking at a data frame for stream #2. There are no flags set and there are 0x22b (555) octets in the frame.

The length of a frame is fairly self-explanatory and the list of possible flags in a data packet is mercifully short. Streams require a bit of mind bending (as we will see). Aside from that, there is little other complexity to data frames in SPDY.

Big whoop, right?

HTTP/1.1 responses are just buffered data. They can be compressed using GZIP compression just fine. So what does SPDY offer here that is a huge win?

Truth be told, SPDY offers nothing over HTTP/1.1 in the response itself. Where SPDY wins, however, and wins big is in the separation of response data from response headers, which SPDY puts into logically separate control frames.

Small, independent data frames are ideal for multiplexing response data. Multiplexing means that response data can be sent back to the browser whenever it is available. In modern HTTP, this is not even remotely possible—interweb tubes are blocked from doing anything else until response has completed. A couple of long running responses can bring web page response time to a crawl.

Conversely, SPDY data packets are emitted by the server as soon as they are ready. Given that modern web pages typically have 75+ additional resources to transfer, the need to multiplex response data is only going to grow.

## 5.1.2. Control Frames

One thing you may have noticed in the data frame is there is no room in there for response and request HTTP headers. I keep saying that SPDY is an HTTP-like protocol so surely it needs to be able to set headers, cookies, sessions and the like. Clients still need to be able to specify that they want HTML returned instead of XML. Servers still need to be able to reply that they are sending PNG images instead of GIFs.

All of this is accomplished with control frames. Control frames are necessarily more complex than data frames. They do change structure *slightly* depending on the type of control frame, but they mostly follow the general form of:

```
 +----------------------------------+
 |C| Version(15bits) | Type(16bits) |
 +----------------------------------+
 | Flags (8)  |  Length (24 bits)   |
 +----------------------------------+
 |               Data               |
 +----------------------------------+
```

The version of SPDY goes into the first 2 octets (always 0x02 at the time of this writing). The type of control frame is specified by a hexadecimal number in the second two octets (at the time of this writing, there are 8 different types).

Flags vary slightly between control frame types. The length of a frame is, again, self-explanatory.

A data frame might look something like:

```
80020001020000 3d00000002000000 01b8f86260862b62062a4252915152520051010a220510b7d
```

Again, it is easier to comprehend when compared with the SPDY spec's ASCII diagram:

```
              +----------------------------------+
```

```
80 02 00 01  |C| Version(15bits) | Type(16bits) |
             +----------------------------------+
02 00 00 3d  | Flags (8)  |  Length (24 bits)   |
             +----------------------------------+
00 00 00 02  |               Data               |
             +----------------------------------+
00 00 00 01
b8 f8 62 60
86 2b 62 06
...
```

As mentioned, the version here is version 2 (0x02) of SPDY (typically written spdy/2). The type of this packet is 0x01, which is a SYN_STREAM. We will become intimately familiar with SYN_STREAMs later. For the time being, it is convenient to think of them as client requests.

We will defer the discussion of the meaning behind this particular flag (teaser: never has 0x02 been so exciting!). The length of this packet is 61 octets (0x3d).

The data section of the frame is type specific. In this case, the first four octets of data are actually more control frame meta-data. Specifically, this is the stream ID for this session. In fact, this is the control frame for the data packet that we just looked at (both stream IDs are 2).

# 5.1.3. This Is Where SPDY Diverges from HTTP

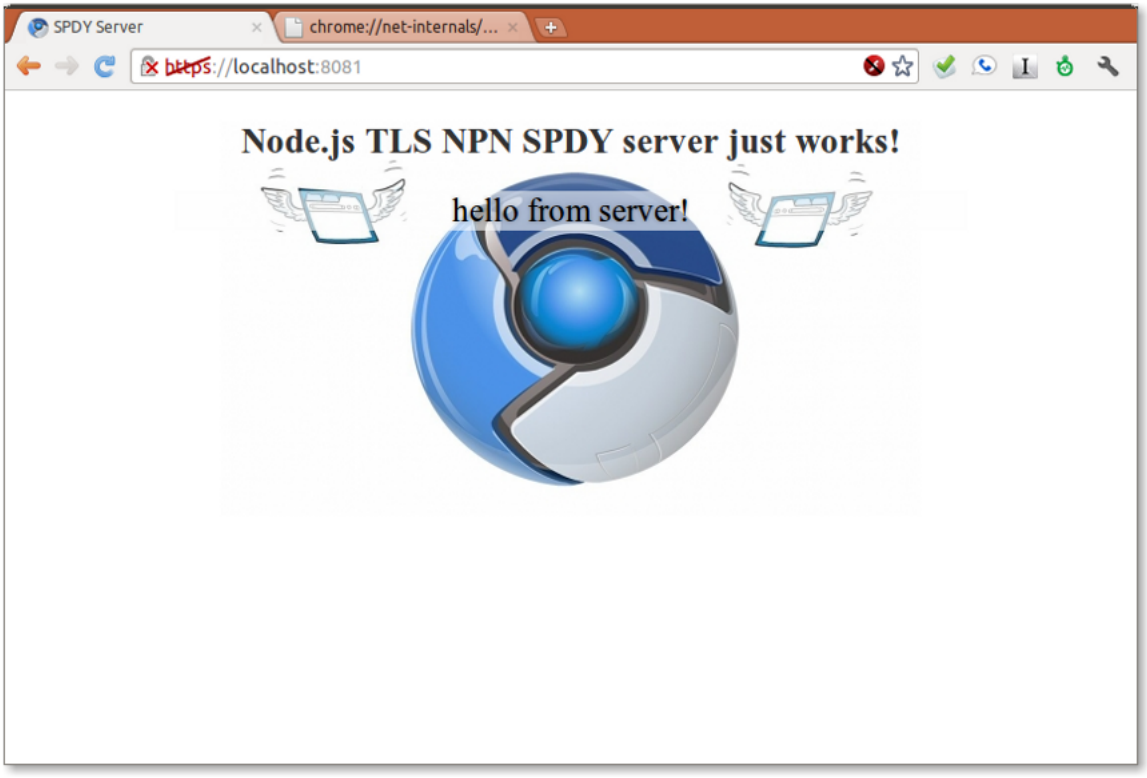
If SPDY were just about shrinking data we might be done here. As we will see in the next chapter, fundamental limitations of HTTP and TCP/IP necessitate that headers and data be separated.

That is, many control frames can be sent before a single data packet is sent. Or a regular control frame can signal a server request (like a normal HTTP request), but the server can then respond with many control frames including a response to the actual request and then data can be sent. The conversation can also be very HTTP-like, with a request followed by a control frame (server headers), followed by data. But in practice, the HTTP-like stuff is quite rare.

# 5.2. Your First SPDY conversation

At the time of this writing, the node-spdy package by Fedor Indutny is the best implementation of SPDY outside of the Google web servers themselves. Included in node-spdy is `test/spdy-server.js` which exercises quite a bit of HTTP-like behavior over SPDY. It is an excellent learning tool so let's start there.

The main page of `test/spdy-server.js` includes HTML, a separate stylesheet, and a background image applied by that stylesheet. As if that were not enough, the web page also POSTs an AJAX request to the server to fill in a `<div>` on that page.




Regardless of whether or not the conversation takes place in plain old HTTP or SPDY, the following occurs in roughly the same order:

1. Browser requests (GET) main page

2. Server responds with HTML
    a. Headers
    b. Data
3. Browser requests (GET) stylesheet
4. Server responds with CSS
    a. Headers
    b. Data
5. Browser requests (GET) background image
6. Server responds with JPEG
    a. Headers
    b. Data
7. Browser POSTs to server for `<div>` data
8. Server responds with plain text
    a. Headers
    b. Data

HTTP keeps the response headers and data logically together. As we will see, in SPDY they are very much separate entities.

**Teaser:** We will find in the next chapter that SPDY can cut back significantly on the back and forth above.

### Note

The packets shown in this section and subsequent chapters were captures with a combination of Wireshark and Google Chrome.

Actual packet contents were captured with Wireshark. Since SPDY is built atop Secure Socket Layers (SSL), some configuration is required to use Wireshark to sniff SSL data. See Appendix B for details on how to use Wireshark to accomplish this.

Chrome is usually better for watching SPDY conversations because it translates binary data into human readable format. More information on Google Chrome tools is available in Appendix C.

The SPDY master will need to hone his or her skills on both.

# 5.2.1. Preamble: SSL Handshake

Prior to anything in SPDY, the normal SSL handshake occurs. Much more information on the particulars of SSL as it relates to SPDY is available in Chapter 8, *SPDY and SSL*. For our purposes here, it is enough to assume that, once the SSL handshake is completed, both client and server are ready for a SPDY session.

# 5.2.2. The Request (SYN_STREAM)

All SPDY sessions start with a SYN_STREAM control frame. Per the spec, SYN_STREAMs have the following profile:

```
+----------------------------------+
|1|  SPDY Version    |       1     |
+----------------------------------+
| Flags (8)  |  Length (24 bits)   |
+----------------------------------+
|X|          Stream-ID (31bits)    |
+----------------------------------+
|X|Associated-To-Stream-ID (31bits)|
+----------------------------------+
| Pri | Unused    |                |
+------------------                |
|     Name/value header block      |
|             ...                  |
```

Each line of the ASCII representation of a SPDY frame is 32 bits wide, or four octets. The only explanation required for the SYN_STREAM ASCII representation is for the first row:

```
+---------------------------------+
|1|  SPDY Version    |       1     |
+---------------------------------+
```

The `1` at the beginning is the control bit. Control frames like SYN_STREAM always have the control bit on. The remainder of the first two octets are the SPDY version, which is 2 at the time of this writing. This makes the first two octets look like `1000 0000 0000 0010`, or in hex, `80 02`. The next two octets represent the type of control frame. SYN_STREAMs are type `1`, meaning that the next two octets look like `00 01`.

The first request for a SPDY page looks something like this when lined up next to the ASCII representation:

```
             +---------------------------------+
80 02 00 01  |1|   SPDY Version   |       1     |
             +---------------------------------+
01 00 01 41  | Flags (8)  |  Length (24 bits)   |
             +---------------------------------+
00 00 00 01  |X|          Stream-ID (31bits)    |
             +---------------------------------+
00 00 00 00  |X|Associated-To-Stream-ID (31bits)|
             +---------------------------------+
00 00 38 ea  | Pri | Unused    |               |
             +-----------------               |
df a2 51 b2  |     Name/value header block      |
62 e0 65 60  |             ...                  |
83 a4 17 06
```

As expected, the first four octets in the SYN_STREAM are `80 02 00 01`.

The next 4 octets indicate that there are no flags set on this SYN_STREAM and that the length of the frame (after and not including the length field) is 0x141 (321) octets:

```
             +---------------------------------+
01 00 01 41  | Flags (8)  |  Length (24 bits)   |
             +---------------------------------+
```

Next up is the stream ID, which shows that this is the first stream sent in this SPDY session:

```
             +---------------------------------+
00 00 00 01  |X|          Stream-ID (31bits)    |
             +---------------------------------+
```

The stream IDs are set by the initiator, in this case the browser. Since this is the very first contact that the browser has had during this session, the ID is `1`. If the session is closed (e.g. the user closes the browser tab or the server shuts down), subsequent stream IDs would again start at `1`, indicating a fresh session. But during the session, as we will see, the stream IDs always increase.

We will not need to worry about the next 4 octets, the associated stream ID, until we delve into the details of SPDY server push. Under normal circumstances, this is always `0`.

The remainder of the packet indicates that the frame has normal priority:

```
...
             +----------------------------------+
00 00 38 ea  | Pri | Unused    |                |
             +------------------                |
df a2 51 b2  |     Name/value header block      |
62 e0 65 60  |             ...                  |
83 a4 17 06
...
```

The name/value header block is compressed with zlib DEFLATE. It is complicated enough that it warrants its own chapter (Chapter 7, *Control Frames in Depth*). DEFLATEd, the packet looks like:

```
'\u0000\r\u0000\u0006accept\u0000?text/html,application/xhtml+xml,application/x
(X11; Linux x86_64) AppleWebKit/534.36 (KHTML, like Gecko)
Chrome/13.0.767.1 Safari/534.36\u0000\u0007version\u0000\bHTTP/1.1
```

If you squint, you can probably make out HTTP request headers in there. In fact, that is the primary purpose of a SYN_STREAM: it initiates a SPDY stream, usually requesting an HTTP resource.

The SYN_STREAM is summarized nicely by the SPDY tab in Chrome's about:net-internals (see Appendix C for details on how to use it):

```
SPDY_SESSION_SYN_STREAM
--> flags = 1
--> accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
    accept-charset: ISO-8859-1,utf-8;q=0.7,*;q=0.3
    accept-encoding: gzip,deflate,sdch
    accept-language: en-US,en;q=0.8
```

```
    host: localhost:8081
    method: GET
    scheme: https
    url: /
    user-agent: Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/535.1 (KHTML, like
    version: HTTP/1.1
--> id = 1
```

So did we really just spend two pages describing an HTTP request in SPDY? Yup. That is exactly what we did.

There are other things that you can do with SYN_STREAMs, but they are best thought of as an HTTP request—especially if you are making the transition from HTTP-land.

So what is next? An HTTP response?

# 5.2.3. The Response (SYN_REPLY)

SYN_REPLYs are the second control frame defined by SPDY. In ASCII representation, they look like:

```
+----------------------------------+
|1|   SPDY Version  |        2     |
+----------------------------------+
| Flags (8)  |  Length (24 bits)   |
+----------------------------------+
|X|          Stream-ID (31bits)    |
+----------------------------------+
| Unused        |                  |
+----------------                  |
|     Name/value header block      |
|              ...                 |
```

Comparing them to SYN_STREAMs, they are nearly identical aside from the type (3rd and 4th octets), which are `2` instead of `1`. The absence of 4 octets for the associated stream ID are the only significant structural difference with SYN_STREAMs. The similarities are intentional to make parsing all control frames as simple as possible.

That leaves intent as the major difference between control frame types. The intent of a SYN_STREAM is to initiate a request. The intent of a SYN_REPLY is to reply to that SYN_STREAM.

Following along with our example site, the SYN_REPLY that comes back from the server looks like:

```
             +----------------------------------+
80 02 00 02  |1|   SPDY Version  |        2     |
             +----------------------------------+
00 00 00 a3  | Flags (8)  |  Length (24 bits)   |
             +----------------------------------+
00 00 00 01  |X|          Stream-ID (31bits)    |
             +----------------------------------+
00 00 78 bb  | Unused        |                  |
df a2 51 b2  +----------------                  |
62 e0 64 e0  |     Name/value header block      |
42 c4 10 03  |              ...                 |
57 76 6a 6a
...
```

As with the SYN_STREAM, the first four octets, `80 02 00 02` are the control bit, the version of SPDY being used, and the type of control frame (0x02 == SYN_REPLY).

The flags are empty and the length is again the length of the frame following (and not including the length field).

The stream ID is still `1`, meaning that this is part of the same stream that was initiated by the SYN_STREAM. Coming from HTTP land it will seem odd, at first, that the SYN_REPLY needs to include the stream ID. After all, this is a response to the request that just came in, what other stream would it be associated with?

But recall, that one of the benefits of SPDY is that it is asynchronous. HTTP requests block until the response comes back from the server. Nothing of the sort happens with SPDY. After the browser sends the SYN_STREAM, it is perfectly valid to send a second SYN_STREAM before any response comes back. It is also valid for the server to initiate its own SYN_STREAMs to push data back to the browser (see Chapter 6, *Server Push—A Killer Feature*).

With all of these packets flying back and forth, we are clearly no longer in HTTP land. We need the stream ID in replies (and other control frames) to keep things straight.

Following the stream ID comes the name/value headers. Looking at Chrome's SPDY tab, the headers are:

```
SPDY_SESSION_SYN_REPLY
--> flags = 0
--> accept-ranges: bytes
    cache-control: public, max-age=0
    connection: keep-alive
    content-length: 698
    content-type: text/html; charset=UTF-8
    etag: "698-1309318367000"
    last-modified: Wed, 29 Jun 2011 03:32:47 GMT
    status: 200 OK
    version: HTTP/1.1
--> id = 1
```

Again, that looks very much like an HTTP/1.1 response to an HTTP/1.1 request.

Seriously? Is there really that much of a difference between SPDY and HTTP?

But wait a second, where is the data…?

# 5.2.4. Server Response: the Data

The next packet that comes back from the server is a data frame as evidenced by the lack of a control bit at the beginning of the frame:

```
             +----------------------------------+
00 00 00 01  |C|       Stream-ID (31bits)       |
             +----------------------------------+
00 00 02 ba  | Flags (8)  |  Length (24 bits)   |
             +----------------------------------+
3c 21 44 4f  |               Data               |
43 54 59 50  +----------------------------------+
45 20 68 74
6d 6c 3e 0a
3c 68 74 6d 6c 3e 0a 20   E html>. <html>.
20 3c 68 65 61 64 3e 0a  20 20 20 20 3c 6c 69 6e    <head>.     <lin
6b 20 72 65 6c 3d 69 63  6f 6e 20 74 79 70 65 3d   k rel=ic on type=
69 6d 61 67 65 2f 70 6e  67 20 68 72 65 66 3d 22   image/pn g href="
66 61 76 69 63 6f 6e 2e  70 6e 67 22 20 2f 3e 0a   favicon. png" />.
...
```

There is no control bit at the beginning of the data frame. There is only one kind of data frame and the version of SPDY makes no difference to a data frame.

Thus, the first four octets, `00 00 00 01` represent the stream ID. Once this data is sent back from the server, the browser will be able to recognize it as the data in response to its initial request (just as it already did with the SYN_REPLY).

There are no flags in this data packet and the length of the data frame (after the length field) is 0x2ba, or 698 octets. The first octets of data are plain old ASCII data: 0x3c = <, 0x21 = !, 0x44 = D, 0x4f = O, 0x43 = C, 0x54 = T, 0x59 = Y, 0x50 = P, 0x45 = E, …. # "<!DOCTYPE…" the opening of an uncompressed HTML page.

In the SPDY tab, this frame is represented as:

```
SPDY_SESSION_RECV_DATA
--> flags = 0
--> size = 698
--> stream_id = 1
```

So is that it for the SPDY equivalent to HTTP's request/response?

# 5.2.5. Server Response: the End of the Data

In SPDY, streams need to be explicitly closed. The client side of stream #1 was closed immediately on sending the request. Chrome sent the SYN_STREAM with a data FIN flag set. But we have yet to close the server side of the stream.

The next frame, another data frame, takes care of that for us:

```
             +----------------------------------+
00 00 00 01  |C|       Stream-ID (31bits)       |
             +----------------------------------+
01 00 00 00  | Flags (8)  |  Length (24 bits)   |
             +----------------------------------+
             |               Data               |
             +----------------------------------+
```

This is confirmed by the SPDY tab:

```
SPDY_SESSION_RECV_DATA
--> flags = 0
--> size = 0
--> stream_id = 1
```

There is no actual data in this packet. Nor should there be. The length field is set to `0`, meaning that there are zero octets of data following the length field.

So the stream is closed. Is that it?

# 5.2.6. Request #2: CSS (SYN_STREAM)

Now that the requested web page has been successfully sent back to the browser, the browser is able to parse the page to see that it needs to request a CSS stylesheet.

So the browser initiates another SYN_STREAM:

```
             +----------------------------------+
80 02 00 01  |1|   SPDY Version   |       1     |
             +----------------------------------+
01 00 00 40  | Flags (8)  |  Length (24 bits)   |
             +----------------------------------+
00 00 00 03  |X|          Stream-ID (31bits)    |
             +----------------------------------+
00 00 00 00  |X|Associated-To-Stream-ID (31bits)|
             +----------------------------------+
40 00 62 e0  | Pri | Unused    |                |
83 c7 a8 10  +------------------                |
38 46 93 8b  |     Name/value header block      |
8b e1 91 64  |             ...                  |
....
```

The fields are used just like they were in the original SYN_STREAM that requested the web page.

But wait.. why is the stream ID `3` instead of `2`? Am I hiding a stream from you? Of course not, would I really do something like that?

What is going on here is that all stream IDs from connections initiated on the client side need to be odd. Even stream IDs are reserved for streams initiated on the server side.

Since neither side knows for sure when the other side might start a new stream, SPDY makes no attempt to figure out how to increment the stream ID across the client and server.

The SPDY tab reports:

```
SPDY_SESSION_SYN_STREAM
--> flags = 1
--> accept: text/css,*/*;q=0.1
    accept-charset: ISO-8859-1,utf-8;q=0.7,*;q=0.3
    accept-encoding: gzip,deflate,sdch
    accept-language: en-US,en;q=0.8
    host: localhost:8081
    method: GET
    referer: https://localhost:8081/
    scheme: https
    url: /style.css
    user-agent: Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/535.1 (KHTML, like
    version: HTTP/1.1
--> id = 3
```

Again, these are normal HTTP headers transported over SPDY, this time requesting `/style.css` from the server.

# 5.2.7. Server Response: CSS Data

In response to the request SYN_STREAM, the server sends back the HTTP headers on a SYN_REPLY:

```
             +----------------------------------+
80 02 00 02  |1|   SPDY Version  |        2     |
             +----------------------------------+
00 00 00 22  | Flags (8)  |  Length (24 bits)   |
             +----------------------------------+
00 00 00 03  |X|          Stream-ID (31bits)    |
             +----------------------------------+
00 00 22 c5  | Unused        |                  |
             +---------------                   |
eb c6 26 e6  |     Name/value header block      |
34 f3 3a d0  |              ...                 |
6c fc 5e 17
07 7b 3d b9
b8 98 5a 3e
07 00 00 00
ff ff
```

That is actually the entire SYN_REPLY. The first SYN_REPLY had 0xa3 (163) octets of data in it. This one only has 0x22 (34) octets. The difference is not the actual size of the headers. Rather it is the zlib compression kicking in.

The results in the SPDY tab:

```
SPDY_SESSION_SYN_REPLY
--> flags = 0
--> accept-ranges: bytes
    cache-control: public, max-age=0
    connection: keep-alive
    content-length: 347
    content-type: text/css; charset=UTF-8
    etag: "347-1309318367000"
    last-modified: Wed, 29 Jun 2011 03:32:47 GMT
    status: 200 OK
    version: HTTP/1.1
--> id = 3
```

# 5.2.8. Server Response: CSS Data

Next up come the two data packets for the CSS:

```
             +----------------------------------+
00 00 00 03  |C|       Stream-ID (31bits)       |
             +----------------------------------+
00 00 01 5b  | Flags (8)  |  Length (24 bits)   |
             +----------------------------------+
62 6f 64 79  |               Data               |
20 7b 0a 20  +----------------------------------+
...
```

And the data FIN that ends the CSS stream:

```
             +----------------------------------+
 00 00 00 05 |C|       Stream-ID (31bits)       |
             +----------------------------------+
 01 00 00 00 | Flags (8)  |  Length (24 bits)   |
             +----------------------------------+
             |               Data               |
             +----------------------------------+
```

The SPDY tab reports normal data received:

```
SPDY_SESSION_RECV_DATA
--> flags = 0
--> size = 347
--> stream_id = 3
SPDY_SESSION_RECV_DATA
--> flags = 0
--> size = 0
--> stream_id = 3
```

# 5.2.9. A POST in SPDY

After the CSS, it might seem as if SPDY has given up all of its tricks and secrets. Make a request (SYN_STREAM), send a response (SYN_REPLY, data frame, data frame with FIN set). But we are only just beginning to uncover all of the cool stuff possible with SPDY.

In this case, the interesting bit comes in the very next packet sent from the browser:

```
             +----------------------------------+
80 02 00 01  |1|   SPDY Version   |       1      |
             +----------------------------------+
00 00 00 46  | Flags (8)  |  Length (24 bits)   |
             +----------------------------------+
00 00 00 05  |X|          Stream-ID (31bits)     |
             +----------------------------------+
00 00 00 00  |X|Associated-To-Stream-ID (31bits)|
             +----------------------------------+
80 00 62 10  | Pri | Unused    |                |
             +------------------                |
84 c7 27 33  |     Name/value header block      |
30 1a 07 59  |             ...                  |
04 f2 a1 96
...
```

Did you spot it? There are actually two bits of interest in that packet.

The more obvious of the two is the presence of the Priority field. This is Chrome telling the server, "Hey! I've thrown a bunch of stuff your way, but don't worry about that other stuff and do this first." The SPDY spec does not strictly enforce the Priority field. In fact,

it says that servers should make a "good faith" effort at working on such pages first. In the nascent SPDY backend market, few if any servers bother to actually do anything in the presence of such a flag.

So the Priority flag is one point of interest in that frame… what's the other?

The other is notable more for its absence that anything else:

```
             +----------------------------------+
00 00 00 46  | Flags (8)  |  Length (24 bits)   |
             +----------------------------------+
```

Notice that there are no flags set here. In the previous two SYN_STREAM frames, the flags were set to `0x01`:

```
             +----------------------------------+
01 00 01 41  | Flags (8)  |  Length (24 bits)   |
             +----------------------------------+
```

Recall that the `0x01` flag in a SYN_STREAM means that the SYN_STREAM was the last thing sent (on this side) of the stream.

So the absence of the flag means that more stuff is to come, but what?

Well, the title of this section gives away that particular surprise, the browser still needs to send along POST data.

In the SPDY tab log, the header very clearly includes a POST:

```
SPDY_SESSION_SYN_STREAM
--> flags = 0
--> accept: */*
    accept-charset: ISO-8859-1,utf-8;q=0.7,*;q=0.3
    accept-encoding: gzip,deflate,sdch
    accept-language: en-US,en;q=0.8
    content-length: 18
    content-type: application/xml
    host: localhost:8081
    method: POST
    origin: https://localhost:8081
    referer: https://localhost:8081/
```

```
    scheme: https
    url: /
    user-agent: Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/535.1 (KHTML, like
    version: HTTP/1.1
--> id = 5
```

Looking at the other headers in there, we can see that we are POSTing to the server root (the homepage). The sample node-spdy server handles homepage GETs and POSTs differently. For the former, the server simply returns the homepage. For the latter, the server echoes the data supplied back to the client. Silly in normal circumstances, but a very effective learning device.

# 5.2.10. POST Data in SPDY

All of that begs the question: how can we POST data in SPDY? In vanilla HTTP, the POST data would be in the body of the HTTP request. There is no body in a SYN_STREAM, so where does the data go?

The absence of the FIN flag on the SYN_STREAM is a strong hint. If the client has not closed its side of the stream, then it has more data to send along.

But what data? Sure, we could check the sample app source code to find out, but it is much more fun packet sniffing!

In this case, the SPDY tab is of minimal help:

```
SPDY_SESSION_RECV_DATA
--> flags = 0
--> size = 18
--> stream_id = 5
SPDY_SESSION_RECV_DATA
--> flags = 0
--> size = 0
--> stream_id = 5
```

Two data frames are sent, the second containing the delayed FIN. The first contains 18 octets of data, but we cannot see the actual contents.

For that, we need Wireshark:

```
             +----------------------------------+
00 00 00 05  |C|       Stream-ID (31bits)       |
             +----------------------------------+
00 00 00 12  | Flags (8)  |  Length (24 bits)   |
             +----------------------------------+
68 65 6c 6c  |               Data               |
6f 20 66 72  +----------------------------------+
6f 6d 20 73
65 72 76 65
72 21
```

Do not be fooled by length of 12. That is `0x12`, which is decimal 18 just like the SPDY tab reports.

But what of the data itself? `0x68`, `0x65`, `0x6c` are ASCII character codes. To see which ones, we can ask Wireshark to dump the printable ASCII along with the hex dump:

```
0000  00 00 00 05 01 00 00 12  68 65 6c 6c 6f 20 66 72   ........ hello fr
0010  6f 6d 20 73 65 72 76 65  72 21                     om serve r!
```

`hello from server!` — hunh? That is being sent from the browser!

This turns out to be a bit of sleight of hand on the part of the test node-spdy server. POSTs to the homepage, which this is, are echoed back to the browser. So POSTing this to the same server should instruct the sample server to echo back `hello from server!` in reply.

Let's take a look at that next.

## 5.2.10.1. Responding to POSTs

In the SPDY tab, the response comes back as a SYN_REPLY, just as did the responses to GET requests:

```
SPDY_SESSION_SYN_REPLY
--> flags = 0
--> connection: keep-alive
    status: 200 OK
    version: HTTP/1.1
--> id = 5
```

Following quickly thereafter, comes the data from the server:

```
SPDY_SESSION_RECV_DATA
--> flags = 0
--> size = 18
--> stream_id = 5
SPDY_SESSION_RECV_DATA
--> flags = 0
--> size = 0
--> stream_id = 5
```

That length of 18 octets is suspiciously familiar. Sniffing the packet, we find:

```
0000   00 00 00 05 00 00 00 12 68 65 6c 6c 6f 20 66 72  ........hello fr
0010   6f 6d 20 73 65 72 76 65 72 21                    om server!
```

The server is saying "hello" after all—but only because the client made it!

# 5.2.11. Request/Response: Image

At this point, the SYN_STREAM, SYN_REPLY, Data, Data FIN patterns should be familiar, so I won't expand the binary session that requests the background images used on the web page. The stream, in the SPDY tab, is what we should expect:

```
SPDY_SESSION_SYN_STREAM
--> flags = 1
--> accept: */*
    accept-charset: ISO-8859-1,utf-8;q=0.7,*;q=0.3
    accept-encoding: gzip,deflate,sdch
    accept-language: en-US,en;q=0.8
    host: localhost:8081
    method: GET
    referer: https://localhost:8081/
    scheme: https
    url: /spdy.jpg
    user-agent: Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/535.1 (KHTML, like
    version: HTTP/1.1
--> id = 7
SPDY_SESSION_SYN_REPLY
--> flags = 0
--> accept-ranges: bytes
    cache-control: public, max-age=0
    connection: keep-alive
    content-length: 76617
```

```
    content-type: image/jpeg
    etag: "76617-1308274555000"
    last-modified: Fri, 17 Jun 2011 01:35:55 GMT
    status: 200 OK
    version: HTTP/1.1
--> id = 7
```

For some reason, the SPDY tab reports large data transfers in very small segments:

```
SPDY_SESSION_RECV_DATA
--> flags = 0
--> size = 8577
--> stream_id = 7
SPDY_SESSION_RECV_DATA
--> flags = 0
--> size = 8217
--> stream_id = 7

# 8 Frames omitted

SPDY_SESSION_RECV_DATA
--> flags = 0
--> size = 0
--> stream_id = 7
```

The number of frames do not seem to correspond to actual frame size.

## 5.2.12. Because We Must: a Favicon

Last up in our SPDY session is the ubiquitous favicon:

```
SPDY_SESSION_SYN_STREAM
--> flags = 1
--> accept: */*
    accept-charset: ISO-8859-1,utf-8;q=0.7,*;q=0.3
    accept-encoding: gzip,deflate,sdch
    accept-language: en-US,en;q=0.8
    host: localhost:8081
    method: GET
    scheme: https
    url: /favicon.png
    user-agent: Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/535.1 (KHTML, like
```

```
    version: HTTP/1.1
--> id = 9
SPDY_SESSION_SYN_REPLY
--> flags = 0
--> accept-ranges: bytes
    cache-control: public, max-age=0
    connection: keep-alive
    content-length: 2582
    content-type: image/png
    etag: "2582-1308274555000"
    last-modified: Fri, 17 Jun 2011 01:35:55 GMT
    status: 200 OK
    version: HTTP/1.1
--> id = 9
SPDY_SESSION_RECV_DATA
--> flags = 0
--> size = 2582
--> stream_id = 9
SPDY_SESSION_RECV_DATA
--> flags = 0
--> size = 0
--> stream_id = 9
```

By now, that should all be old news: `SYN_STREAM -> SYN_REPLY + DATA + DATA_FIN`.

# 5.3. Summary

Phew! That was a whirlwind tour of SPDY. Hopefully after this chapter, you have a feel for what SPDY does and why. You should also be getting a feel for the data and what it looks like in hexadecimal format and in the SPDY tab.

From a logical perspective, not much has changed between vanilla HTTP and SPDY. But always keep in mind that the SPDY conversations are taking place on a single pipe. That pipe is warm and the data transferred is making full use of the bandwidth between you and the server without contention with other pipes to the same site.

That is all well and good, but, by now, you must be itching for some asynchronous goodness. What can SPDY do when you start pushing it? We will start answering that in the next chapter…

# Chapter 6. Server Push—A Killer Feature

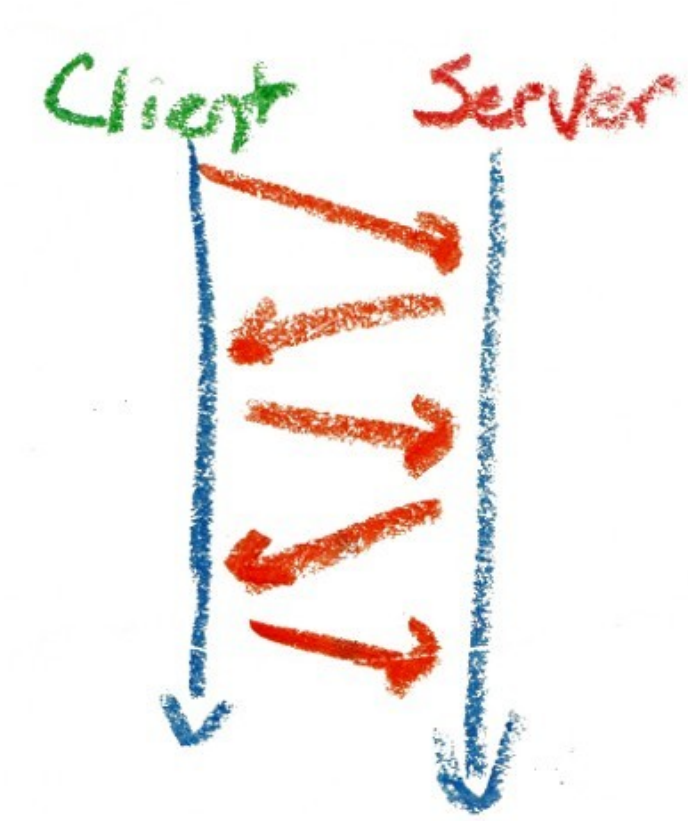
Server push in SPDY means something very different than what web developers have meant for years. It has nothing to do with

- Comet
- AJAX Long Polling
- Websockets

Rather think back to the sole purpose of SPDY — to decrease page load time. The longer the throbber throbs, the more likely a potential client is to hit the Back button.

With that in mind, SPDY server push means to push additional responses along with the normal response to a request.

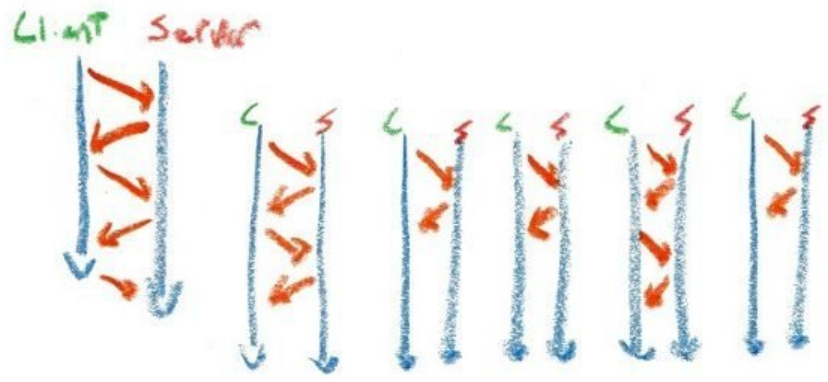
In vanilla HTTP, a request for a web page might look something like:




The web browser makes a request of a web server, which responds with a web page. Based on the contents of that web page, the browser will have to make subsequent requests of the server to ask for stylesheets, images, *etc.*.

If browsers only had a single pipe on which to talk to the server, the conversation would take a very long time. As described in Chapter 5, *The Basics*, HTTP does not allow asynchronous conversation, so the browser has to wait for a response before beginning a new request.

All modern browsers get around this by initiating up to 6 pipes to the web server:



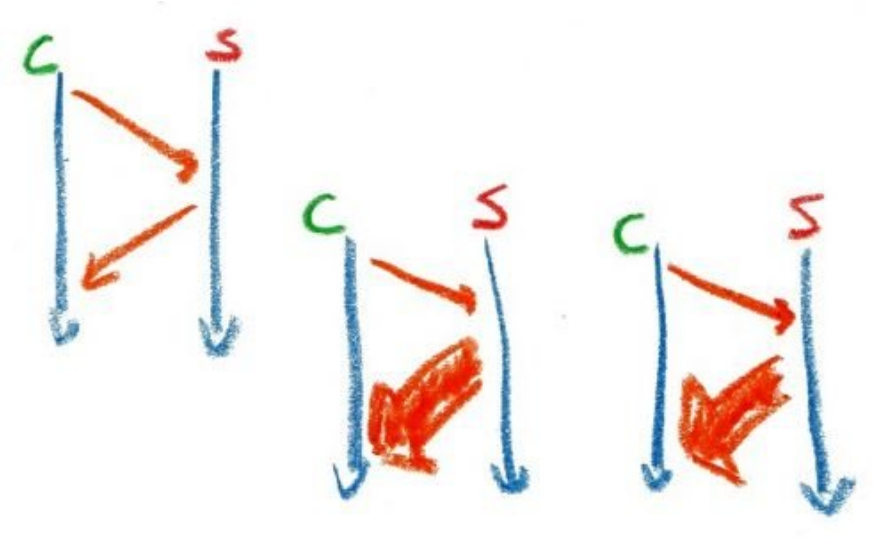
Each of those pipes still incurs the limitation of synchronous request/response pairs. If a web page has more than 6 additional resources that need to be loaded (typical modern sites can have 40+), then the browser will still be waiting a significant amount of time for round-trip latency on each of these pipes.

Additionally, as discussed in Chapter 5, *The Basics*, each of these pipes has to go through the same warm up period as the TCP/IP CWND is negotiated. The more pipes, the more likely there will be contention between the pipes, and the more likely it is that a network error will impact one of them.

Lastly, note that the secondary requests cannot begin until at least some of the web page response has come through. All modern web browsers are capable of making secondary requests once they have read the <head> section of a page, and know what CSS and Javascript will be needed. Even so, that does not begin until the response is at least underway.

To get around some of these limitations, web sites have, in recent years, begun to package multiple stylesheets into a single larger stylesheet. It is possible to do the same with Javascript files and images (the latter in the form of sprites).

Asset packaging makes HTTP conversations look more like:

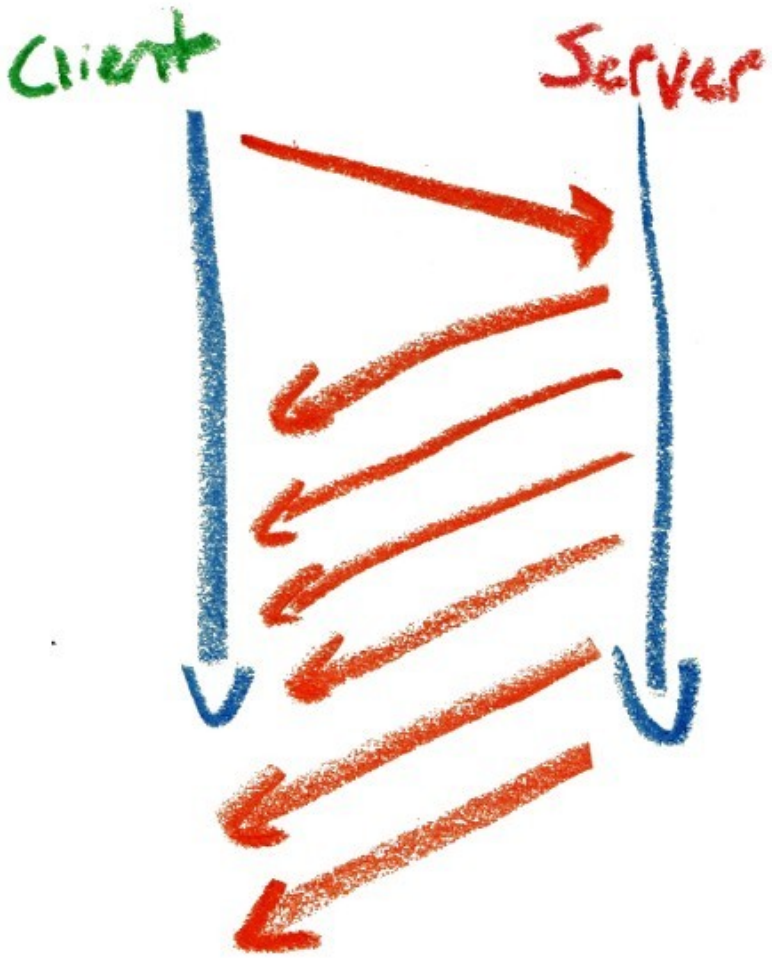



An obvious deficiency with this approach is that the browser still needs to wait for the first response before it can know what to request subsequently.

Also problematic with asset packaging is additional maintenance. Asset packaging requires non-trivial amounts of configuration and management to ensure that these resources are combined and served properly.

# 6.1. SPDY Server Push

SPDY solves this problem by pushing additional resources along with the requested resource. If a browser requests a site’s homepage, SPDY provides the capability to push along all of the images, stylesheets and Javascript files at the same time:




As we found in Chapter 5, *The Basics*, SPDY is an asynchronous protocol with all conversation taking place on a single pipe. What server push illustrates is that SPDY conversations are not even required to be bi-directional.

## Caution

There is the possibility of a race condition if the web page is returned before the pushed resources arrive at the browser. In that case, the browser will request these

resources even as the server is delivering them. Given this, it is strongly suggested that the server at least initiate server push for all resources before sending along the data on a SYN_REPLY.

Under normal HTTP, if a browser has previously downloaded a resource, it keeps a copy of that resource in the local cache. The next time the user goes to the same page, the browser can serve data from the local cache rather than incurring the same round-trip downloads from the server.

SPDY server push uses this mechanism by pushing resources directly into the browser's cache. Once the browser determines that it needs a stylesheet for the current page, it finds that the stylesheet is already in cache—even if this is the first time that a user has visited the page!

# 6.2. A Detailed Look at Server Push

With the background out of the way, let's take a look at a detailed SPDY server push conversation. Just as in previous chapters, the binary protocol will be displayed in octets next to an ASCII legend of the bit.

**URLs**

An absolute URL includes the scheme (https), the hostname (example.com), the port (8081), the path to the resource (`/about/me.html`), path information (`#top`), and the query string (`?style=fabulous`). A typical absolute URL looks like `https://example.com/about/me.html`.

A root relative URL, the kind most used in SPDY contains just the path to the resource starting after the hostname in an absolute URL. An example of a root relative URL is `/about/me.html`.

As we will see, the most important part of the server push is the URL of the resource being pushed out to the browser. Unlike other URLs in SPDY conversations, it must be an absolute URL.

**Note**

This is a sample SPDY server push conversation. As we will discuss later, the order of the packets can be re-arranged.

First up in a server push is the client request…

# 6.2.1. Client Request (SYN_STREAM)

The initial SYN_STREAM is relatively simple. It has the control bit set (the very first bit of the packet) because this is a control frame (i.e. a frame that does not contain actual data). The current SPDY version (as of the writing of this book) is 2. And since this is a SYN_STREAM, the Type is 0x01.

```
             +----------------------------------+
80 02 00 01  |1|   Version        |     Type    |
             +----------------------------------+
00 00 01 38  | Flags (8)  |  Length (24 bits)   |
             +----------------------------------+
00 00 00 01  |X|          Stream-ID (31bits)    |
             +----------------------------------+
00 00 00 00  |X|Associated-To-Stream-ID (31bits)|
             +----------------------------------+
00 00 62 60  | Pri | Unused    |                |
             +------------------                |
64 60 06 05  |     Name/value header block      |
81 42 46 49  |             ...                  |
49 41 b1 95
...
```

Since this is the first packet sent by the browser, the stream ID is 1. The name/value header block contains "normal" HTTP headers, but are compressed to save space. When uncompressed, they might look something like:

```
...
\u0000
\u000e
accept-charset
\u0000
\u001e
ISO-8859-1,utf-8;q=0.7,*;q=0.3
```

```
...
\u0000
\u0004
host
\u0000
\u000e
localhost:8081
\u0000
\u0006
method
\u0000
\u0003
GET
\u0000
\u0007
version
\u0000
\u0008
HTTP/1.1
...
```

Per the SPDY way, the header is packed as tightly as possible, names and values introduced by the size of each and separated by null characters. The entire buffer is then compressed using zlib compression.

Once the server receives this request, it is able to recognize the resource needed and reply accordingly…

## 6.2.2. Server Response (SYN_REPLY)

The server response packet looks very much like the request. A SYN_REPLY is primarily distinguished from the initiating SYN_STREAM by the Type value, which is 0x02 for a SYN_REPLY.

```
             +----------------------------------+
80 02 00 02  |1|   Version       |      Type    |
             +----------------------------------+
00 00 00 a4  | Flags (8)  |  Length (24 bits)   |
             +----------------------------------+
00 00 00 01  |X|          Stream-ID (31bits)    |
             +----------------------------------+
```

```
00 00 78 bb  | Unused        |                  |
df a2 51 b2  +---------------                  |
62 e0 64 e0  |     Name/value header block     |
42 c4 10 03  |              ...                |
57 76 6a 6a
...
```

The SYN_REPLY retains the stream ID of the SYN_STREAM that initiated the connection. Aside from that, the SYN_REPLY is very similar to the SYN_STREAM to which it is responding.

The decompressed contents of the name/value block, should be familiar to anyone that has ever viewed an HTTP response:

```
\u0000
\t
\u0000
\n
connection
\u0000
\n
keep-alive
\u0000
\u000e
content-length
\u0000
\u0003
698
\u0000
\r
cache-control
\u0000
\u0011
public, max-age=0
\u0000
\r
last-modified
\u0000
\u001d
Sat, 14 May 2011 02:36:14 GMT
\u0000
\u0004
etag
```

```
\u0000
\u0013
"698-1305340574000"
\u0000
\f
content-type
\u0000
\u0018
text/html; charset=UTF-8
\u0000
\r
accept-ranges
\u0000
\u0005
bytes
\u0000
\u0006
status
\u0000
\u0006
200 OK
\u0000
\u0007
version
\u0000\bHTTP/1.1
```

But no data has been sent yet! This is just a handshake with some optional headers describing the reply itself (none of the to-be-pushed resources).

## 6.2.3. Server Push (SYN_STREAM)

Next comes the first hint of server push. This is another control frame (as evidenced by the control bit at the beginning of the frame), but is not a SYN_REPLY. Rather, this is a new stream, a new SYN_STREAM, being initiated by the server.

```
             +----------------------------------+
80 02 00 01  |1|   Version        |     Type    |
             +----------------------------------+
02 00 00 51  | Flags (8)  |  Length (24 bits)   |
             +----------------------------------+
00 00 00 02  |X|          Stream-ID (31bits)    |
             +----------------------------------+
```

```
00 00 00 01  |X|Associated-To-Stream-ID (31bits)|
             +-----------------------------------+
00 00 62 60  | Pri | Unused    |                 |
             +------------------                 |
             |     Name/value header block       |
23 c2 37 cc  |             ...                   |
a0 40 52 c8
28 29 29 28
...
```

The type of this stream is 0x01 (the fourth octet), which indicates that it is a SYN_STREAM. The most interesting stuff in this frame, however, follows the type.

This is the first packet that we have seen with flags set. In this case, 0x02 indicates FLAG_UNIDIRECTIONAL. Most streams in SPDY are bi-directional, meaning that both client and server can send data. Uni-directional streams are only one way, meaning that only the server (the initiator of this stream) can send data.

The next field in this frame is the stream ID. It is *not* `1` like the SYN_STREAM or SYN_REPLY. This is, after all, a new stream. Of significance is that the stream ID is 2. This is not because it comes immediately after the already open stream ID #1. All client initiated streams have IDs that are odd. Since this is a server initiated stream, it needs to be even.

Last up in the SYN_STREAM frame is the associated stream ID, which links this reply back to the original SYN_STREAM.

In the name/value header block, only three things are required:

`status`
: `200` the HTTP status code with which the server would reply if this were a normal GET. This value should always be 200.

`version`
: The HTTP version being used underneath SPDY. This should always be "HTTP/1.1".

`url`
: The **fully-qualified** URL of the resource. Although most URLs in name/value header blocks are root relative (`/about/me.html`), this URL must be fully-qualified (`https://example.com/about/me.html`).

If this SYN_STREAM is used to push out a stylesheet for the page requested, then another SYN_STREAM is required to send out images…

# 6.2.4. Server Push #2 (SYN_STREAM)

Again, this is a control frame of type 0x01 (SYN_STREAM).

```
              +----------------------------------+
80 02 00 01   |1|   Version        |     Type    |
              +----------------------------------+
02 00 01 5b   | Flags (8)  |  Length (24 bits)   |
              +----------------------------------+
00 00 00 04   |X|          Stream-ID (31bits)    |
              +----------------------------------+
00 00 00 01   |X|Associated-To-Stream-ID (31bits)|
              +----------------------------------+
00 00 62 60   | Pri | Unused    |                |
              +------------------                |
              |     Name/value header block      |
23 c2 37 cc   |             ...                  |
a0 40 52 c8
28 29 29 28
...
```

The uni-directional flag is set and the associated stream ID is still #1. Of significance in this frame is the stream ID. As mentioned when discussing the first push frame, all server initiated stream IDs are even. Hence, the ID of this second stream must be 4.

At this point, the request has been handled, the initial response (SYN_REPLY) has been sent back to the browser having two new server push streams (both with new, unique stream IDs). But no data has been sent.

Let’s have a look at that next.

# 6.2.5. Data Push

Data frames in SPDY are set apart from control frames by the control bit. The very first bit, on in a control frame, is off for a data frame:

```
              +----------------------------------+
```

```
00 00 00 02  |C|       Stream-ID (31bits)       |
             +----------------------------------+
00 00 02 2b  | Flags (8)  |  Length (24 bits)   |
             +----------------------------------+
62 6f 64 79  |               Data               |
20 7b 0a 20  +----------------------------------+
20 70 61 64
64 69 6e 67
....
```

There is not too much to a SPDY data frame and this is no exception. The stream ID is followed by flags (none here) and the length of the packet. After a scant 8 octets of introduction, the actual data follows.

After all of the data has been sent, a FIN (final) frame is sent:

```
             +----------------------------------+
00 00 00 02  |C|       Stream-ID (31bits)       |
             +----------------------------------+
01 00 00 00  | Flags (8)  |  Length (24 bits)   |
             +----------------------------------+
             |               Data               |
             +----------------------------------+
```

Again this is for stream ID #2, but this frame is empty save for a flag of 0x01, which is a DATA_FIN flag.

# 6.2.6. Data Push #2

There is nothing new in the data for stream ID #4. Aside from the stream ID, everything behaves just as it did for the initial data push:

- Data frames are sent with 8 octets of introduction, followed by the actual data.
- The last data frame is empty save for a flag of 0x01, DATA_FIN.

# 6.2.7. Response Data Reply

And *finally* we come to the reply to the original request. The actual data frame containing the requested web page:

```
             +---------------------------------+
00 00 00 01  |C|       Stream-ID (31bits)      |
             +---------------------------------+
00 00 02 ba  | Flags (8)  |  Length (24 bits)  |
             +---------------------------------+
3c 21 44 4f  |               Data              |
43 54 59 50  +---------------------------------+
45 20 68 74
6d 6c 3e 0a
...
```

### Tip

0x3c = <, 0x21 = !, 0x44 = D, 0x4f = O, 0x43 = C, 0x54 = T, 0x59 = Y, 0x50 = P, 0x45 = E, …. Put the characters together and you get: "<!DOCTYPE…"—the opening of an uncompressed HTML page.

Once the web page data has been sent, the last bit of housekeeping is to FIN the original request stream:

```
             +---------------------------------+
00 00 00 01  |C|       Stream-ID (31bits)      |
             +---------------------------------+
01 00 00 00  | Flags (8)  |  Length (24 bits)  |
             +---------------------------------+
             |               Data              |
             +---------------------------------+
```

Again this is for stream ID #1, but this frame is empty save for a flag of 0x01, which is a DATA_FIN flag.

### Important

The FIN frame for the original request stream is the point at which new push streams are no longer valid.

At this point, additional server push, *for this stream* is no longer possible.

# 6.3. Summary

During the SPDY server push conversation, a total of 7 logically distinct frames were sent in both directions between client and server:

1. Request
2. Response — Headers Only
3. Push Headers #1
4. Push Headers #2
5. Push Data #1
    a. Push Data #1 FIN
6. Push Data #2
    a. Push Data #2 FIN
7. Response Data
    a. Response Data FIN

## 6.3.1. Practical Ordering

As noted in the Response Data Reply section, once a FIN is sent for the original request stream, no more push streams can be initiated. The implication is that more push streams can be sent in between the response data and the response data FIN.

### Caution

All resources that will be needed to satisfy the original request (e.g. CSS, Javascript, images) **must** be initiated *before* the Response Data is sent. If the server replies with the requested web page, and then initiates a push of the page's CSS, a

race condition has been created. The browser will parse the web page and request the CSS even as the server is trying to push it.

Always send the headers of pushed data prior to responding to the original request.

Why might you want to initiate new push streams after the original response has been handled?

The most obvious use case is to ensure that the most requested pages on your site are already in the user's cache. Browser vendors have been trying to do this for years with `<link rel=...>` and the `next`, `prev` attributes. No one ever uses these tags because they are silly. The client will issue multiple requests after the original request has been fulfilled forcing the server to use resources to respond. The server has no way of prioritizing such requests over direct user requests.

With SPDY server push, the original request can be fulfilled. There can even be additional streams to request resources not covered by the server push. After all that, the server can *still* push resources—as long as they have been initiated via SYN_STREAM before the original request is FINed.

The upshot is that the server can give priority to servicing real requests and push out these secondary pages when it has a spare cycle. This is trivial with asynchronous frameworks.

A more current use case for post-request data push is the increased practice of using client-side view templates. Prior to SPDY server push, servers would need to push an entire site (albeit packaged and compressed) to respond to the initial request. But a SPDY server can send just enough for the initial template to be rendered. The remaining resources can be pushed at the server's leisure.

# 6.4. Conclusion

As we saw back in Chapter 4, *SPDY Push*, SPDY server push is relatively easy to implement at an application level. We should now understand how the SPDY protocol supports this very powerful feature. We will revisit it yet again in Chapter 10, *The Future*. First, we will have a brief look at the remaining control frames supported by SPDY.

# Chapter 7. Control Frames in Depth

In Chapter 5, *The Basics* and Chapter 6, *Server Push—A Killer Feature*, we saw several examples of control frames. We were also introduced to the concept of separation of control and data frames and some of the power inherent in such an approach. It is worth now, taking a detailed look at the various control frames in SPDY.

SPDY controls frames operate at about the same level as HTTP headers. So let's start by looking at HTTP headers and how they map into SPDY. You might be wondering how HTTP headers have any real impact on website speed. When most web developers think of HTTP headers, they are little more than this:

```
$ curl -i google.com
HTTP/1.1 301 Moved Permanently
Location: http://www.google.com/
Content-Type: text/html; charset=UTF-8
Date: Sat, 09 Jul 2011 19:58:17 GMT
Expires: Mon, 08 Aug 2011 19:58:17 GMT
Cache-Control: public, max-age=2592000
Server: gws
Content-Length: 219
X-XSS-Protection: 1; mode=block
```

HTTP headers are tiny little things, right? Tiny but certainly important. They hold URLs, content-type, cache information and HTTP statuses. How could something that small impact the performance of websites where the content being transported is on the order of tens or even hundreds of kilobytes?

To answer that, consider a request for a "real" resource (not one that redirects):

```
$ curl -iv www.google.com
> GET / HTTP/1.1
> User-Agent: curl/7.21.3 (x86_64-pc-linux-gnu) \
    libcurl/7.21.3 OpenSSL/0.9.8o zlib/1.2.3.4 \
    libidn/1.18
> Host: www.google.com
> Accept: */*
>
```

```
HTTP/1.1 200 OK
Date: Sat, 09 Jul 2011 18:54:33 GMT
Expires: -1
Cache-Control: private, max-age=0
Content-Type: text/html; charset=ISO-8859-1
Set-Cookie: PREF=ID=e09213fd384211a1:FF=0:\
  TM=1310237673:LM=1310237673:S=Sk14a4iaRQqjt-fP; \
  expires=Mon, 08-Jul-2013 18:54:33 GMT; \
  path=/; \
  domain=.google.com
Set-Cookie: NID=48=WlUjEKinL00iPD5LI3zCb_\
  SEp4srx970f06DtQTIvJhpslYioB4QlPpdhoG3qK-\
  A65ygFiS2IApzTX_aNa1GNvvO-ASI73UVs7j10LB1PYxRkb\
  VyQjRdKCOFBcH5fYOT; \
  expires=Sun, 08-Jan-2012 18:54:33 GMT; \
  path=/; \
  domain=.google.com; \
  HttpOnly
Server: gws
X-XSS-Protection: 1; mode=block
Transfer-Encoding: chunked
```

New in that request are cookies and a transfer-encoding. Web application developers are no doubt aware of additional headers like these. We rarely pay them much heed, though. Sure there are headers like Transfer-Encoding and `X` headers like `X-XSS-Protection`. But they do not take up much space. Right?

And cookies famously cannot be larger than 4 KB. And why should they be? As web application developers, we are aware that cookies occupy some bandwidth so we are not about to shove hundreds of kilobytes in there. That is, after all, what web pages are for!

Regardless, those are only a couple hundred bytes added in the request. They certainly are not going to make or break a high performance web site.

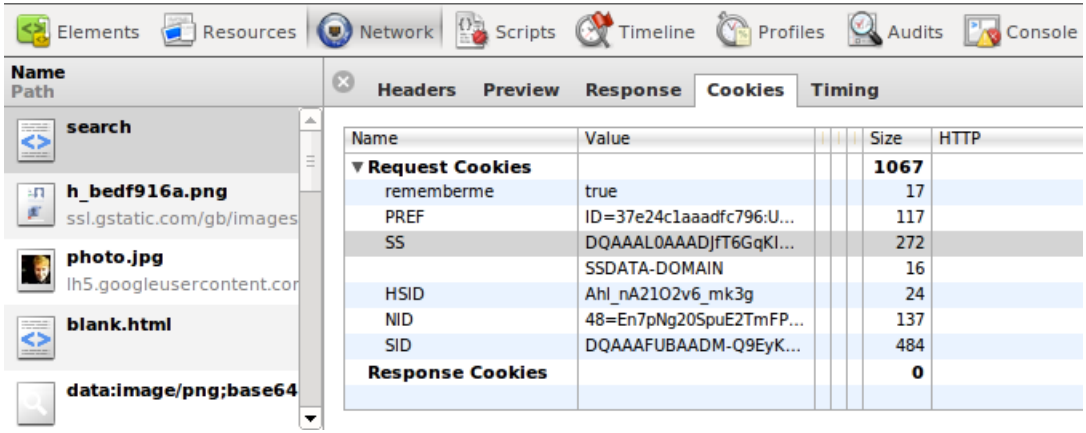
Actually, they do. Consider a subsequent Google search:

```
$ curl -A Mozilla -iv 'www.google.com/search?q=spdy' \
  -b 'PREF=ID=1690849646215d90:FF=0:\
      TM=1310237866:LM=1310237866:\
      S=S-CE38CpgHCqFLmM; \
      NID=48=hw5jrA2T8PYPTZeCyJD31tTXImZVGepEmC\
```

```
       8h5HDFlXeRN5ZzvKsmrMS2bqQn0aslMxZcfSTqt3L\
       RmJ2PwXfG75AHZktKt6e3LcenvfFyfq64fizpp0yd\
       Pn7nLa0ubXKQ'
> GET /search?q=spdy HTTP/1.1
> User-Agent: Mozilla
> Host: www.google.com
> Accept: */*
> Cookie: PREF=ID=1690849646215d90:FF=0:\
    TM=1310237866:LM=1310237866:\
    S=S-CE38CpgHCqFLmM; \
    NID=48=hw5jrA2T8PYPTZeCyJD31tTXImZVGepEmC\
     8h5HDFlXeRN5ZzvKsmrMS2bqQn0aslMxZcfSTqt3L\
     RmJ2PwXfG75AHZktKt6e3LcenvfFyfq64fizpp0yd\
     Pn7nLa0ubXKQ

HTTP/1.1 200 OK
Date: Sat, 09 Jul 2011 19:01:23 GMT
Expires: -1
Cache-Control: private, max-age=0
Content-Type: text/html; charset=ISO-8859-1
Set-Cookie: PREF=ID=1690849646215d90:\
  U=4fce8698c6a6f594:FF=0:TM=1310237866:\
  LM=1310238083:S=iDIY_ao-QSlIIwo7; \
  expires=Mon, 08-Jul-2013 19:01:23 GMT; \
  path=/; \
  domain=.google.com
Server: gws
X-XSS-Protection: 1; mode=block
Transfer-Encoding: chunked
```

As we can see, the cookies are sent by both the browser and server. Each time a request is made, the cookies are sent. Each time a response comes back, the cookies are sent. Every single time.

Even a bandwidth conscious company like Google is seeing 225 bytes worth of cookies go back and forth on each request. The amount of bandwidth only increases as sessions are authenticated and authorized:




Unless your company has a Director of Cookies (mmmm… that's going to be *my* next job title), your applications probably set cookies that are even larger. And yet, it is only around 5 kilobytes that might be in the headers. So what?

The problem, of course, is that the headers go back **and** forth for *every* request—not just the dynamic web pages that need them. Every stylesheet on your site, every tiny image, every Javascript resource sends these headers back and forth.

If the typical web site had an additional 40+ resources associated with each page, **that** can add up. 40 requests and 40 responses each with 5 KB cookies in them ends up being 400 kilobytes! Worst of all is that many of the headers (especially the big ones) are redundant.

Certainly there are steps that one can take with vanilla HTTP to mitigate this situation. You could hire that Director of Cookies (did I mention I'm available?). You might even try installing a company-wide policy to limit cookie size.

A tried-and-true method is the Content Distribution Network (CDN). If all of your static data is hosted in a separate domain, browser and server will never send application cookie data back and forth. Cookies are limited to the server that set them in the first place. If application server and CDN are separate, then most requests associated with a web page will not have header bloat.

The drawback to CDNs is cost and added deployment complexity.

Personally, I recommend the Director of Cookies, but however you do it, there are definite steps that you can take to mitigate header bloat. But wouldn't it be nice if HTTP just did that for us?

HTTP might not, but SPDY does…

# 7.1. SPDY Headers

SPDY does not do away with cookies or any other HTTP headers. They are present and repeated just as often as in vanilla HTTP. What SPDY offers over HTTP is extremely aggressive, but still quite fast compression of those headers.

SPDY uses zlib to perform compression of headers before transport. The zlib library comes from the same group that gave us the ubiquitous gzip. Although not quite as famous as its progenitor, zlib is in very heavy usage in Unix systems (like Linux and Mac OSX), as well as various gaming systems.

The zlib compression algorithm offers exactly what is needed for networking: significant compression with minimal overhead.

Most of us tend to think of compression as a one-time operation. We zip up our files and send them off in an email. Or we tar/gzip an application before deploying it to production. In fact, zlib allows for the compression stream to remain open as additional data is compressed. This further minimizes resources required over the lifetime of a compression session. Exploiting this is a key factor in SPDY's ability to transport data quickly.

To further boost its compression ability, SPDY employs a lookup dictionary. The dictionary includes common HTTP headers and values:

```
optionsgetheadpostputdeletetraceacceptaccept-char
setaccept-encodingaccept-languageauthorizationexp
ectfromhostif-modified-sinceif-matchif-none-match
if-rangeif-unmodifiedsincemax-forwardsproxy-autho
rizationrangerefererteuser-agent10010120020120220
320420520630030130230330430530630740040140240340 4
40540640740840941041141241341441541641750050150250
03504505accept-rangesageetaglocationproxy-authent
icatepublicretry-afterservervarywarningwww-authen
```

```
ticateallowcontent-basecontent-encodingcache-cont
rolconnectiondatetrailertransfer-encodingupgradev
iawarningcontent-languagecontent-lengthcontent-lo
cationcontent-md5content-rangecontent-typeetagexp
ireslast-modifiedset-cookieMondayTuesdayWednesday
ThursdayFridaySaturdaySundayJanFebMarAprMayJunJul
AugSepOctNovDecchunkedtext/htmlimage/pngimage/jpg
image/gifapplication/xmlapplication/xhtmltext/pla
inpublicmax-agecharset=iso-8859-1utf-8gzipdeflate
HTTP/1.1statusversionurl
```

A dictionary allows the compression algorithm to further optimize compressed data. It does so by referencing a dictionary offset and length where possible. In addition to better compression, this also requires less computation time than employing a compression algorithm.

As you might imagine, there is trade-off between a large dictionary and performance. Looking through the SPDY dictionary, web developers and network admins should each recognize nearly all of the strings in there. This provides assurance that the SPDY spec does a good job of identifying the most common HTTP headers.

**Note**

Data compression, which is expected to be deprecated in a future spec, does *not* use a dictionary, but is still per stream.

As we have seen in previous chapters, both client and server need to set headers in SPDY sessions. SPDY mandates that both sides of the conversation have their own zlib sessions for compressing data. In practice, this results in a slight hit in bandwidth for the first two packets on either side of the conversation. Subsequent packets, however, see a remarkable compression.

But don’t take my word for it, let’s examine some headers in detail!

# 7.1.1. Name/Value Header Blocks

When we looked at SYN_REPLYs in previous chapters, we represented them like this:

```
+----------------------------------+
```

```
|1|  SPDY version   |        2     |
+---------------------------------+
| Flags (8)  |  Length (24 bits)  |
+---------------------------------+
|X|          Stream-ID (31bits)   |
+---------------------------------+
| Unused        |                 |
+---------------                  |
|     Name/value header block     |
|              ...                |
```

We looked in detail at the top part of these frames and saw, as if by magic, that the headers came from that name/value block looking something like:

```
0x09
...
0x00 0x00 0x00 0x0c
content-type
0x00 0x00 0x00 0x18
text/html; charset=UTF-8
0x00 0x00 0x00 0x06
status
0x00 0x00 0x00 0x06
200 OK
...
```

Hexadecimal aside, we can see normal HTTP headers in there. What is going in here is that name/value blocks are actually packed in more like this:

```
+-----------------------------------+
| Number of Name/Value pairs (int32) |
+-----------------------------------+
|     Length of name (int32)        |
+-----------------------------------+
|           Name (string)           |
+-----------------------------------+
|     Length of value  (int32)      |
+-----------------------------------+
|          Value   (string)         |
+-----------------------------------+
|           (repeats)
```

If we match headers against the ASCII diagram, we find:

```
                        +-----------------------------------+
0x09                    | Number of Name/Value pairs (int32) |
                        +-----------------------------------+
0x00 0x00 0x00 0x0c     |     Length of name (int32)         |
                        +-----------------------------------+
content-type            |           Name (string)            |
                        +-----------------------------------+
0x00 0x00 0x00 0x18     |     Length of value  (int32)       |
                        +-----------------------------------+
text/html; charset=UTF-8 |          Value   (string)          |
                        +-----------------------------------+
0x00 0x00 0x00 0x06     |     Length of name (int32)         |
                        +-----------------------------------+
status                  |           Name (string)            |
                        +-----------------------------------+
0x00 0x00 0x00 0x06     |     Length of value  (int32)       |
                        +-----------------------------------+
200 OK                  |          Value   (string)          |
                        +-----------------------------------+
...                     |           (repeats)
                        -----
```

Breaking it down, there are 9 total name/value pairs in this particular frame:

```
                        +-----------------------------------+
0x09                    | Number of Name/Value pairs (int32) |
                        +-----------------------------------+
...
```

The length of the first header name, "content-type", is 12 (`0x0c`) octets:

```
...
                        +-----------------------------------+
0x00 0x00 0x00 0x0c     |     Length of name (int32)         |
                        +-----------------------------------+
content-type            |           Name (string)            |
                        +-----------------------------------+
...
```

The value associated with "content-type" is "text/html; charset=UTF-8", which is 24 (`0x18`) characters long:

```
...
```

```
                          +-----------------------------------+
0x00 0x00 0x00 0x18       |     Length of value  (int32)      |
                          +-----------------------------------+
text/html; charset=UTF-8  |          Value   (string)         |
                          +-----------------------------------+
...
```

Each of the name/value pairs in the remainder of the header block follow along using this same scheme. Given that headers in HTTP are of variable size and order, SPDY really has no choice but to push the headers together in this fashion. One could argue that the length of the names will never require 32 bits—4,294,967,295 characters. And likely the values would never reach 4 GB in length either (not with a fast protocol anyway). But saving 2 octets, which would take the headers down to `int16`, can hold a maximum of 65,535 characters. That is rather large for a header, but it is not out of the realm of possibility that a larger string might be needed.

SPDY could have gone the HTTP route and separated everything with a sequence of bytes (e.g. carriage return + new line). But reading known lengths from a buffer is far quicker than parsing every character in a long string, looking for special sequences so that additional processing can be done.

Besides, zlib will make up for the slight bloat. In spades…

# 7.1.2. Decompression with Zlib

SPDY uses separate zlib compression streams on both sides of a SPDY session. This means that regardless of whether or not you are working on the client or server side of things, you *will* need to work with zlib. In some programming languages that can be more challenging than in others (Ruby, I'm looking at you!). Regardless of which language you use, the general approach remains the same.

### Note

This section is rather dense and can be safely skipped unless you are interested in the inner workings of zlib compression.

In the following Python code, we will see how the server would go about decompressing incoming headers. First we create a zlib stream object:

```
z_stream = _z_stream()
```

### Zlib Streams

Zlib stream objects are little more than a C pointer data structure to which zlib can write. The entire structure can be more or less discerned from the follow Python class definition:

```
class _z_stream(C.Structure):
    _fields_ = [
        ("next_in", C.POINTER(C.c_ubyte)),
        ("avail_in", C.c_uint),
        ("total_in", C.c_ulong),
        ("next_out", C.POINTER(C.c_ubyte)),
        ("avail_out", C.c_uint),
        ("total_out", C.c_ulong),
        ("msg", C.c_char_p),
        ("state", C.c_void_p),
        ("zalloc", C.c_void_p),
        ("zfree", C.c_void_p),
        ("opaque", C.c_void_p),
        ("data_type", C.c_int),
        ("adler", C.c_ulong),
        ("reserved", C.c_ulong),
    ]
```

The only parts that SPDY-ists need to worry about are the `next_in`, `avail_in`, and `total_in` attributes (and their `_out` counterparts). The remaining attributes are only used by zlib internally.

Getting decompression underway involves a call to one of zlib's inflate methods, here `inflateInit2_`:

```
_zlib.inflateInit2_(C.byref(z_stream), MAX_WINDOW_BITS, ZLIB_VERSION, C.sizeof(
```

We are passing in a reference to the zlib stream object in which the decompression will take place. The other values passed in are only of interest to C neckbeards (if you really want to know, they are `15`, `1.2.3.4` and the size of the stream pointer object).

With the stream object prepped for z-streaming action, we need a place to collect the results:

```
# The out-buffer is just a range of memory to store the results
outbuf = C.create_string_buffer(CHUNK)
```

And we need to give the z-stream the input buffer (the compressed data) and the output buffers:

```
z_stream.avail_in  = \
  len(data1)
z_stream.next_in   = \
  C.cast(C.c_char_p(data1), C.POINTER(C.c_ubyte))
z_stream.avail_out = \
  CHUNK
z_stream.next_out  = \
  C.cast(outbuf, C.POINTER(C.c_ubyte))
```

Wow. That was a lot of setup. We are *still* not quite done.

The dictionary needs to be assigned to the stream. Assigning a zlib dictionary is a bit awkward. You do not simply assign the dictionary. First, you need to attempt to inflate (decompress) the stream. That will return an error code indicating that a dictionary is required. Only **then** can you actually inflate:

```
# Try inflate, it fails because it needs a dictionary
_zlib.inflate(C.byref(z_stream), Z_SYNC_FLUSH)

# Set the dictionary
_zlib.inflateSetDictionary(
  C.byref(z_stream),
  C.cast(C.c_char_p(dictionary), C.POINTER(C.c_ubyte)),
  len(dictionary))

# Inflate for real now that the dictionary is set
_zlib.inflate(C.byref(z_stream), Z_SYNC_FLUSH)
```

So what does this look like in action? The first packet through is going to look something like this:

```
octets_1 = [
0x38,0xea,0xdf,0xa2,0x51,0xb2,0x62,0xe0,0x62,0x60,
0x83,0xa4,0x17,0x06,0x7b,0xb8,0x0b,0x75,0x30,0x2c,
0xd6,0xae,0x40,0x17,0xcd,0xcd,0xb1,0x2e,0xb4,0x35,
0xd0,0xb3,0xd4,0xd1,0xd2,0xd7,0x02,0xb3,0x2c,0x18,
```

```
0xf8,0x50,0x73,0x2c,0x83,0x9c,0x67,0xb0,0x3f,0xd4,
0x3d,0x3a,0x60,0x07,0x81,0xd5,0x99,0xeb,0x40,0xd4,
0x1b,0x33,0xf0,0xa3,0xe5,0x69,0x06,0x41,0x90,0x8b,
0x75,0xa0,0x4e,0xd6,0x29,0x4e,0x49,0xce,0x80,0xab,
0x81,0x25,0x03,0x06,0xbe,0xd4,0x3c,0xdd,0xd0,0x60,
0x9d,0xd4,0x3c,0xa8,0xa5,0x2c,0xa0,0x3c,0xce,0xc0,
0x0f,0x4a,0x08,0x39,0x20,0xa6,0x15,0x30,0xe3,0x19,
0x18,0x30,0xb0,0xe5,0x02,0x0b,0x97,0xfc,0x14,0x06,
0x66,0x77,0xd7,0x10,0x06,0xb6,0x62,0x60,0x7a,0xcc,
0x4d,0x65,0x60,0xcd,0x28,0x29,0x29,0x28,0x66,0x60,
0x06,0x79,0x9c,0x51,0x9f,0x81,0x0b,0x91,0x5b,0x19,
0xd2,0x7d,0xf3,0xab,0x32,0x73,0x72,0x12,0xf5,0x4d,
0xf5,0x0c,0x14,0x34,0x00,0x8a,0x30,0x34,0xb4,0x56,
0xf0,0xc9,0xcc,0x2b,0xad,0x50,0xa8,0xb0,0x30,0x8b,
0x37,0x33,0xd1,0x54,0x70,0x04,0x7a,0x3e,0x35,0x3c,
0x35,0xc9,0x3b,0xb3,0x44,0xdf,0xd4,0xd8,0x44,0xcf,
0x18,0xa8,0xcc,0xdb,0x23,0xc4,0xd7,0x47,0x47,0x21,
0x27,0x33,0x3b,0x55,0xc1,0x3d,0x35,0x39,0x3b,0x5f,
0x53,0xc1,0x39,0x03,0x58,0xec,0xa4,0xea,0x1b,0x1a,
0xe9,0x01,0x7d,0x6a,0x62,0x04,0x52,0x16,0x9c,0x98,
0x96,0x58,0x94,0x09,0xd5,0xc4,0xc0,0x0e,0x0d,0x7c,
0x06,0x0e,0x58,0x9c,0x00,0x00,0x00,0x00,0xff,0xff]
```

Which results in something like:

```
'\x00\x0a
\x00\x06accept\x00\x3ftext/html,application/xhtml+xml,application/xml;q=0.9,*/*
\x00\x0eaccept-charset\x00\x1eISO-8859-1,utf-8;q=0.7,*;q=0.3
\x00\x0faccept-encoding\x00\x11gzip,deflate,sdch
\x00\x0faccept-language\x00\x0een-US,en;q=0.8
\x00\x04host\x00\x0flocalhost:10000
\x00\x06method\x00\x03GET
\x00\x06scheme\x00\x05https
\x00\x03url\x00\x01/
\x00\x0auser-agent\x00\x67Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/534.30 (K
\x00\x07version\x00\x08HTTP/1.1'
```

Now wait a second. That’s the opposite of space saving!! We go through all of that so that 260 octets of "compressed" data can become 388 octets?!

The answer is that we accept a hit on the first packet to realize **huge** savings on subsequent packets. The first packet contains the dictionary initialization from the other side of

the conversation. That dictionary initialization is what makes the packet large and also instructs zlib to fail on the first inflate with a need-dictionary error.

But that is just one packet in hundreds that are going to pass between you and the client. The next packet might contain compressed headers that are as small as:

```
octets_2 = [
0x42,0x8a,0x02,0x66,0x60,0x60,0x0e,0xad,0x60,0xe4,
0xd1,0x4f,0x4b,0x2c,0xcb,0x04,0x66,0x33,0x3d,0x20,
0x31,0x58,0x42,0x14,0x00,0x00,0x00,0xff,0xff]
```

That is only 29 octets of data which expand to:

```
'\x00\0a
\x00\x06accept\x00\x03*/*
\x00\x0eaccept-charset\x00\x1eISO-8859-1,utf-8;q=0.7,*;q=0.3
\x00\x0faccept-encoding\x00\x11gzip,deflate,sdch
\x00\x0faccept-language\x00\x0een-US,en;q=0.8
\x00\x04host\x00\x0flocalhost:10000
\x00\x06method\x00\x03GET
\x00\x06scheme\x00\x05https
\x00\x03url\x00\x0c/favicon.ico
\x00\0auser-agent\x00\x67Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/534.30 (KH
\x00\x07version\x00\x08HTTP/1.1'
```

Nice. 29 octets expand to 341 octets.

That was quite an excursion into the land of compression and low-level Python, but hopefully it was worth it. We were able to see in some detail how zlib compression impacts the second frame sent in a SPDY session. Of course the importance is that the second and *every* other frame has this super compression.

The most common SPDY Control Frames, SYN_SESSION, SYN_REPLY, HEADERS all use compressed name/value pairs, so it is an important concept to understand and be able to work with—especially if you are involved in implementing a SPDY framework.

# 7.2. HEADERS

The HEADERS control frame is not covered elsewhere in this book, but is definitely worth a mention. Sometimes the server may not be ready to send name/value information

in a SYN_REPLY. In HTTP, this would result in blocking until the server is able to respond.

In SPDY, a SYN_REPLY can go out, and the client and server can continue merrily communicating about other resources. Once the server is finally ready to emit the header information that would otherwise have gone out with the SYN_REPLY, it can send it in the well-named HEADERS frame, which looks like:

```
+----------------------------------+
|C|  SPDY version   |      8       |
+----------------------------------+
| Flags (8)  |  Length (24 bits)   |
+----------------------------------+
|X|          Stream-ID (31bits)    |
+----------------------------------+
|  Unused (16 bits) |              |
|--------------------              |
| Name/value header block          |
+----------------------------------+
```

The Stream-ID in the HEADERS packet is the same that began with a SYN_SESSION and continued with a SYN_REPLY that included empty or partial name/value data.

The HEADERS frame can be especially helpful in server push situations. All push streams need to be initialized before the response stream closes. It is nice to be able to establish the push streams without telling the browser anything about the data to be pushed into its cache. Once the server is done sending the response, it can then inspect the files to be pushed and send out content-type, content-length, etc. in a HEADERS frame. In this manner, nothing blocks the sending of the initial response, minimizing browser wait time.

# 7.3. When Things Go Wrong: RST_STREAM

The RST_STREAM control frame is always a fun thing to see in practice. When one side of a SPDY stream is doing something wrong, it can signal the other end to stop with this frame.

I got this a *lot* when I was learning server push:

```
SPDY_SESSION_PUSHED_SYN_STREAM
--> associated_stream = 1
--> flags = 2
--> url: /style.css
--> id = 2
SPDY_SESSION_SEND_RST_STREAM
--> status = 1
--> stream_id = 2
```

In this case, I attempted to push the `/style.css` stylesheet directly into the browser cache. The problem was that I used a relative URL rather than an absolute URL. Since Chrome could not process this control frame, it signaled to the server to stop doing what it was doing.

In the case of the server push, this actually saves a significant amount of time. Since the browser cannot process a resource with an invalid URL, any data sent back by the server would be ignored—it would occupy bandwidth for no reason. The RST_STREAM prevents the server from taking any further action with nothing but a tiny SYN_STREAM.

Eventually, the browser would process the web page, recognize that it needs to load the stylesheet and that the stylesheet is not in cache. When that happens, the browser simply does what it does in HTTP—requests it from the server.

# 7.4. SPDY Settings

The `SETTINGS` control frame is easy to mistake for HTTP headers. Its ASCII representation is:

```
+----------------------------------+
|1| SPDY version |         4       |
+----------------------------------+
| Flags (8)  |  Length (24 bits)   |
+----------------------------------+
|         Number of entries        |
+----------------------------------+
|          ID/Value Pairs          |
|             ...                  |
```

One might expect that the "ID/Value Pairs" can transmit HTTP header information. The give away that they cannot is the "ID" in "ID/Value", which actually represents an integer

ID. Those IDs do not describe SYN_STREAMs or SYN_REPLYs. Rather they describe the connection for the SPDY sessions.

As of this writing, there are only 5 possible values:

- SETTINGS_UPLOAD_BANDWIDTH (`0x01`)
- SETTINGS_DOWNLOAD_BANDWIDTH (`0x02`)
- SETTINGS_ROUND_TRIP_TIME (`0x03`)
- SETTINGS_MAX_CONCURRENT_STREAMS (`0x04`)
- SETTINGS_CURRENT_CWND (`0x05`)

As can be guessed from the names, these settings are specific to the underlying TCP/IP streams. They allow implementations to tune the TCP/IP stream as soon as a connection is established, rather than waiting for TCP/IP slow start to kick in.

An interesting point about these settings is that they persist for the duration of the SPDY session. There is no need to send them along with every request as with HTTP Headers.

At the application layer, there is little of interest in `SETTINGS` control frames. The ability to warm the TCP/IP tubes within the first few packets of a conversation is an exciting capability—if application servers support it (node-spdy does not).

# 7.5. Other Control Frames

For completeness sake, I will briefly mention the remaining control frames.

The WINDOW_UPDATE control frame is used to establish per-stream control flow. It is meant to allow either side of the stream to pause a stream. For example, if one side is processing a lot of data and having a hard time buffering everything, it can signal the sender to pause by sending a zero-sized window.

The PING control frame is similar to a TCP/IP ping. It measures the round trip time of a SPDY session. Since the PING control frame contains no information other than an

ID, it is the smallest control frame defined by SPDY. As such, the PING frame actually measures the minimum round trip time.

The GOAWAY control frame signals the end of a session. This is used in the case that the server is shutting down or the web page is closed.

# 7.6. Wrapping Up Control Frames

Control frames are where all the SPDY action is. We have already spent several chapters discussing two of them: SYN_STREAM and SYN_REPLY. The compression of the name/value pairs in both (also used in HEADERS) is a very fast, high compression algorithm—it would have to be if SPDY is going to include it.

99% of your time in SPDY will be spent in SYN_STREAMs and SYN_REPLYs and their associated data streams. Of the remaining control frames, the RST_STREAM is the most common (and most despised) one to be seen. The remaining control frames will likely grow in use as SPDY and its implementations mature. For now, it is good to be aware of them, but they are likely to be of little practical use.

# Chapter 8. SPDY and SSL

SPDY runs over Secure Socket Layer (SSL) connections. This is what makes SPDY secure by design. There are flags in Google Chrome that allow it to run without SSL, but all sorts of things break if you try it.

SPDY requires SSL. The sooner you embrace it, the better.

Some time is actually lost when loading pages due to the overhead of SSL. If SPDY is all about page load time, isn't this some kind of compromise?

Of course it is.

But seriously, in 2012 and beyond, all sites are going to require SSL. It is an age in which Firesheep and its ilk make session hacking trivial. The plain text web is dead and SPDY is quite up front about being built to last in 2012 and beyond.

## 8.1. SPDY Backwards Compatibility with HTTP/1.1

Using SSL does have some advantages in addition to security. The primary benefit in the context of SPDY is Next Protocol Negotiation (NPN), which is currently a draft standard of the IETF [1].

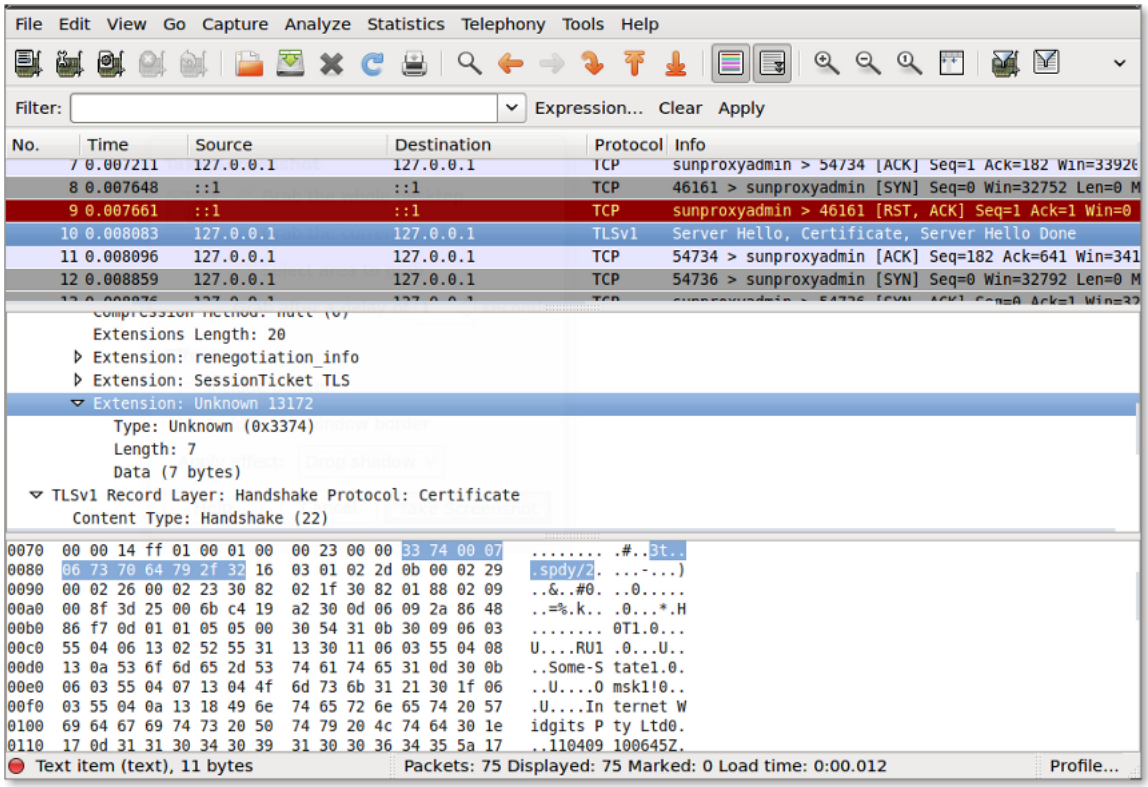
NPN allows a site to publish which protocols it supports during normal SSL negotiation.

**Tip**

Wireshark is an invaluable tool for inspecting SSL sessions. It can even decrypt SSL data if you have access to the server's private keys. More information on using Wireshark with SSL and SPDY is in Appendix B, *Using Wireshark to Sniff SSL Packets*.

[1] http://tools.ietf.org/html/draft-agl-tls-nextprotoneg-02

NPN is relatively new, so even an excellent tool like Wireshark is not quite sure what to make of it. It is part of the Server Hello portion of SSL negotiation. It is extension data near the end of the certificate information:

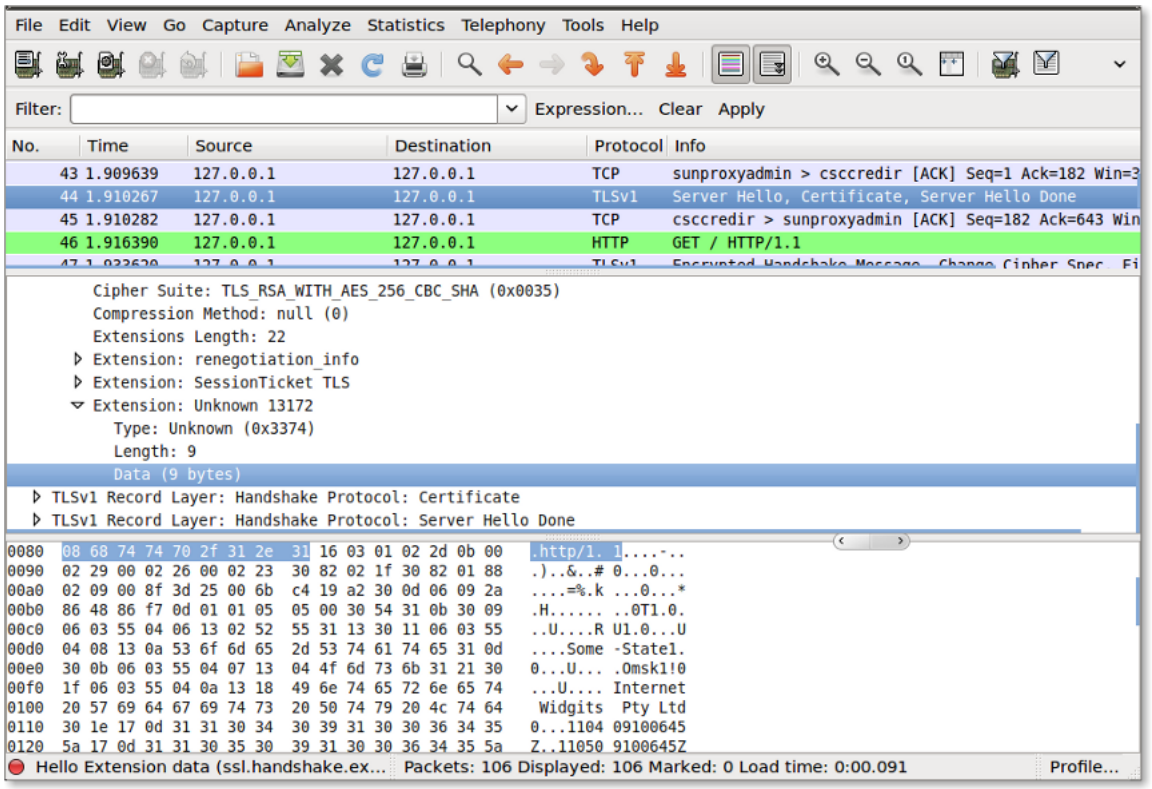



NPN-aware browsers like Chrome use this to make SPDY requests of a site instead of using the default HTTP/1.1.

### Important

Since NPN is so new, most browsers will simply ignore it and make plain-old HTTPS requests of a site. Unless and until SPDY becomes a widespread standard, SPDY-enabled sites will need some kind of normal HTTPS strategy — either redirection or handling it in tandem.

It is possible for a site to advertise that it explicitly supports only HTTP.




In such a case, the browser will interact with the site over HTTPS—even if both server and client could otherwise support SPDY.

# 8.2. Creating SSL Certificates

To create a new certificate on Unix-like systems, use the `openssl` command:

```
openssl genrsa -out key.pem 1024
```

This generates the actual private key that the server will use to establish a secure connection. A private key (and an associated public key) would be sufficient to establish a secure connection. It is not enough to establish a *trusted*, secure connection.

A public/private key combination would ensure that no one can decipher the packets [2] being exchanged between client and server. The browser could very well be exchanging sensitive data with the Russian mob, but at least no one could see it, right?

Well, no, that is generally not acceptable, so a trusted third party is needed to verify a connection. To do that, a certificate signing request is needed:

```
openssl req -new -key key.pem -out request.pem
```

The resulting `request.pem` file contains a certificate request. The certificate request can be submitted to a certificate authority (*e.g.* VeriSign, GoDaddy, Comodo) to be vetted. Certificate authorities have a standard set of hoops through which a website needs to jump before the certificate is "signed" by the certificate authority. Once signed, and installed on a website, the browser now has a mechanism for verifying that the server is who it says it is (and is not the Russian mob).

The signed certificate, along with the original key from `genrsa` are the two pieces of information needed to run a secure website. In the Apache web server, these two are denoted with the `SSLCertificateFile` and `SSLCertificateKeyFile` values:

```
SSLCertificateFile    /etc/ssl/certs/cert.pem
SSLCertificateKeyFile /etc/ssl/private/key.pem
```

How they are configured in SPDY applications depends on the server or programming language being used. All the way back in Chapter 2, *Your First SPDY App*, we saw that express-spdy (and the underlying node-spdy) configure SSL when the express.js server is instantiated:

```
var app = express.createServer({
  key: fs.readFileSync('keys/spdy-key.pem'),
  cert: fs.readFileSync('keys/spdy-cert.pem'),
  ca: fs.readFileSync('keys/spdy-csr.pem'),
  NPNProtocols: ['spdy/2', 'http/1.1']
}
```

Expect something similar no matter which framework you are using.

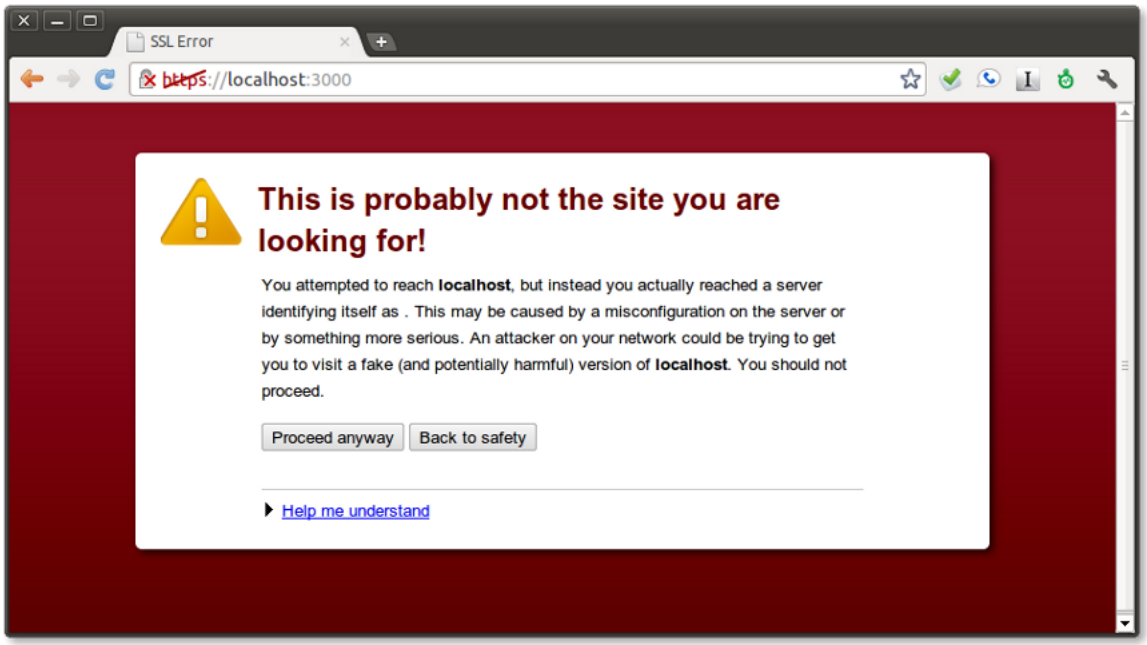
[2] Fine, fine. Maybe not *ensure*, but make prohibitively difficult for third parties to snoop. Satisfied?

# 8.2.1. Impatience

If you are looking to just get up and running locally, it is possible to "self-sign" a certificate:

```
openssl x509 -req -days 30 -in request.pem -signkey key.pem -out cert.pem
```

Browsers will give you all sorts of errors:




Proceeding past these in development mode is pretty much the status quo.

# 8.2.2. Being Your Own CA

If you are OK with scary browser warnings when developing applications or can afford to buy legit certificates, feel free to skip this section.

It takes a bit of effort to be your own Certificate Authority (CA), but it can be a big win. It is especially good to be able to sign your own certificates if you need to obtain detailed timing information on SPDY sessions.

### Warning

This section relies on the `openssl ca` command. Per its own documentation, "the `ca` command is quirky and at times downright unfriendly."

In the `openssl.cnf` config file [3], set the location where the CA work is going to take place:

```
## Where everything is kept
dir             = ./CA
```

This setting is relative to your current working directory. It allows you to test things out in other directories before doing them for real in `$HOME/CA` or wherever you prefer to keep your CA work.

### Important

It is certainly possible to use these instructions to establish yourself as a CA within a company or other private organization. Users would either have to be instructed to manually install your CA certificate or would have to use browsers that come pre-bundled with your CA certificate. If you do something like that, **you must guard your private CA certificate!** Ideally this work would be performed on a secure machine and would need to be done with escalated user privileges. Above all, ensure that the private CA key is not readable by anyone other than the user charged with signing certificates.

As already noted, `ca` is a fussy little command. The first example of this is the need to manually setup the directory that stores everything (we are using `./CA` here). `openssl ca` requires a flat file database to exist. It also needs a pointer to the next certificate ID that will be blessed by our awesome new CA. The exact name of those files is also stored in `openssl.cnf`:

```
database = $dir/index.txt
serial  = $dir/serial
```

[3] I am using a custom openssl installation, which makes it easier to customize without fear of breaking system-level settings. See Section A.2, “Edge openssl” for details.

Before you ask, no, `ca` cannot create these files itself (it's unfriendly, remember?). The database file needs to simply exist. The serial pointer file needs to contain the ID of the first certificate that we will create:

```
$ cd
$ mkdir CA
$ touch CA/index.txt
$ echo '01' > CA/serial
```

Also needed by `ca` are sub-directories to hold new certificates and various "private" information such as the CA's private key:

```
new_certs_dir   = $dir/newcerts
private_key     = $dir/private/cakey.pem
```

`openssl ca` will populate those files, so all that is needed is a simple `mkdir`:

```
$ mkdir CA/newcerts CA/private
```

### Note

Real CAs need to guard that private directory with their very lives.

Before moving on, this is a good time to make our lives a bit easier by setting some useful defaults in `openssl.cnf`:

```
countryName_default  = US
stateOrProvinceName_default = Maryland
0.organizationName_default = EEE Computes, LLC
```

Those values will be used both when creating new certificates and when signing those certificate requests.

With that, it is time to create our CA certificate:

```
$ openssl req -new -x509 \
  -extensions v3_ca \
  -out CA/cacert.pem \
  -keyout CA/private/cakey.pem \
  -days 3650
```

The `out` and `keyout` options store the CA's public certificate and private key respectively. The locations are chosen to match the configuration in `openssl.cnf`:

```
certificate = $dir/cacert.pem
private_key = $dir/private/cakey.pem
```

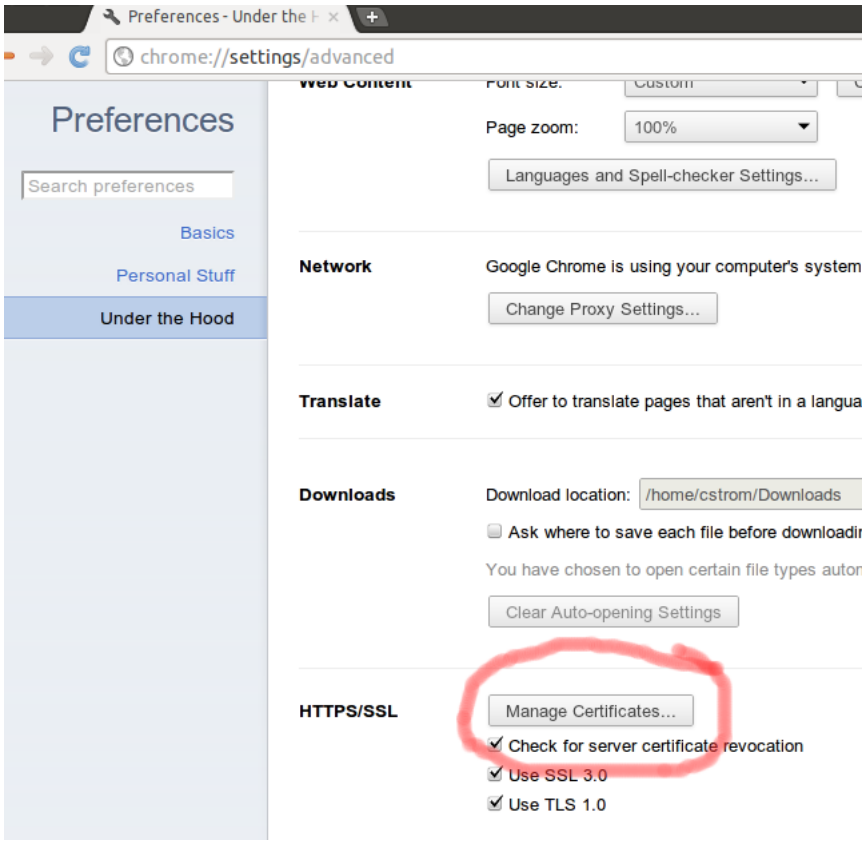
After running `openssl req`, the private key is written to disk, after which you will be prompted to answer some CA questions:

```
Generating a 1024 bit RSA private key
.....................++++++
.....................................................++++++
writing new private key to 'CA/private/cakey.pem'
Enter PEM pass phrase:
Verifying - Enter PEM pass phrase:
  -----
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
  -----
Country Name (2 letter code) [US]:
State or Province Name (full name) [Maryland]:
Locality Name (eg, city) []:
Organization Name (eg, company) [EEE Computes, LLC]:
Organizational Unit Name (eg, section) []:
Common Name (eg, YOUR name) []:EEE Computes Root CA
Email Address []:ca@eeecomputes.com
```

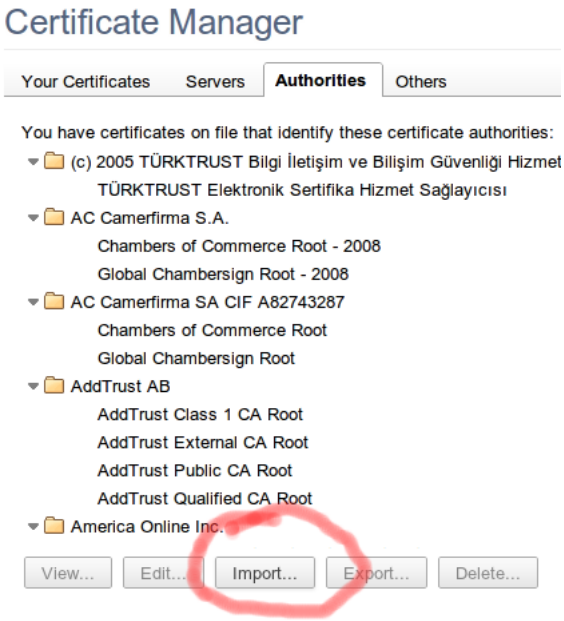
Most of that comes from the default values that we put in the configuration file. The common name is the name that will identify the CA certificate when it is installed on the browser.

That is all there is to it! We now have a private key stored in `CA/private/cakey.pem` and a public certificate located in `CA/cacert.pem`. It is the public certificate, `CA/cacert.pem`, that can be installed in a browser so that the browser will recognize it as a new Certificate Authority.

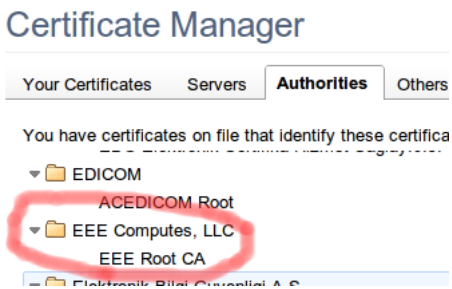
The installation process for new root CAs varies from browser to browser. In Chrome, this is accomplished in Preferences // Under the Hood. First click the "Manage Certificates" button:




On the Root Certificates tab of the resulting dialog, import the new, *public* `cacert.pem` with the "Import" button:




After that, we have ourselves new, official CA:




This certificate may only be valid in our browser, but that is very likely all we want for testing anyway.

At this point, we have a real, live CA, but CAs are useless without certificates to sign.

As we saw earlier, we need to first generate the private key, then issue a certificate signing request. Here are the two operations for a local server named `jaynestown.local` [4]:

[4] mudders love him

```
$ openssl genrsa -out jaynestown.key 1024
$ openssl req -new -key jaynestown.key -out jaynestown.csr
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
  -----
Country Name (2 letter code) [US]:
State or Province Name (full name) [Maryland]:
Locality Name (eg, city) []:
Organization Name (eg, company) [EEE Computes, LLC]:
Organizational Unit Name (eg, section) []:
Common Name (eg, YOUR name) []:jaynestown.local
Email Address []:chris@eeecomputes.com
```

The answer to the `Common Name` question is the one that matters here. Our answer, `jaynestown.local`, specifies the hostname for which this certificate will be valid.

The first output file from the above commands is the `jaynestown.key` private key. If we were sending a certificate request to a "real" CA, we would keep this hidden. The certificate signing request, `jaynestown.csr` is the other output. This needs to be signed by our new CA:

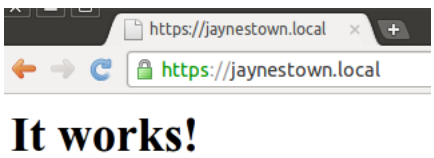
```
openssl ca -out ./CA/newcerts/jaynestown.crt -in jaynestown.csr
Using configuration from /home/cstrom/local/ssl/openssl.cnf
Enter pass phrase for ./CA/private/cakey.pem:
Check that the request matches the signature
Signature ok
Certificate Details:
        Serial Number: 1 (0x1)
        Validity
            Not Before: Jul 27 01:32:00 2011 GMT
            Not After : Jul 26 01:32:00 2012 GMT
        Subject:
            countryName               = US
            stateOrProvinceName       = Maryland
            organizationName          = EEE Computes, LLC
            commonName                = jaynestown.local
            emailAddress              = chris@eeecomputes.com
        X509v3 extensions:
```

```
            X509v3 Basic Constraints:
                CA:FALSE
            Netscape Comment:
                OpenSSL Generated Certificate
            X509v3 Subject Key Identifier:
                1C:21:62:29:B2:BB:84:26:4B:69:93:5D:E8:A2:82:A5:0C:EA:0C:00
            X509v3 Authority Key Identifier:
                keyid:3D:1B:A2:E4:94:D4:0C:D0:3B:D5:BC:78:B9:F7:97:40:73:C8:59:

Certificate is to be certified until Jul 26 01:32:00 2012 GMT (365 days)
Sign the certificate? [y/n]:y


1 out of 1 certificate requests certified, commit? [y/n]y
Write out database with 1 new entries
Data Base Updated
```

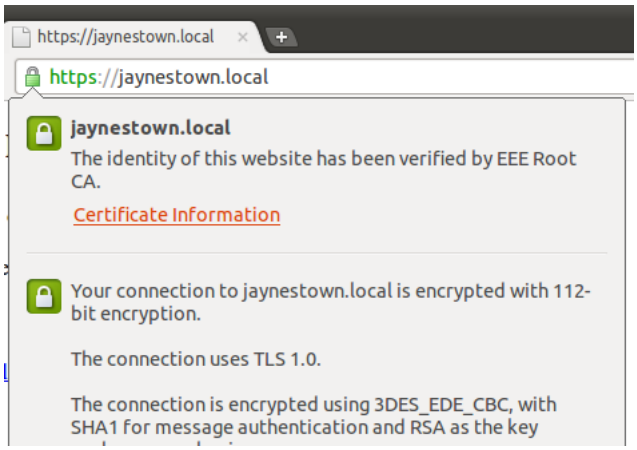
The signed `jaynestown.crt` certificate is now ready for use along with the private `jaynestown.key`. To use these two in Apache, the following configuration might be used:

```
SSLCertificateFile    /etc/ssl/certs/jaynestown.crt
SSLCertificateKeyFile /etc/ssl/private/jaynestown.key
```

And, with that, our local jaynestown server has a legit certificate:




And the certificate is signed by our new Certificate Authority:




Admittedly, that is a significant amount of work. But the ability to generate and sign testing certificates that will not throw scary browser warnings can save immense time when working with SPDY.

## 8.2.3. Certificates That Play Nice with Wireshark

There are times that you will need to troubleshoot SPDY connections and look at the packets themselves. Since SPDY runs entirely on SSL, this may seem an impossible task. After all, SSL is supposed to prevent third parties from viewing the contents of the connection.

Well, it turns out that Wireshark *can* decipher SSL connections. It does so the same way that a web server does it—with the server's private key.

### Important

For the love of all that is holy, please be extremely careful doing anything with a private key. If a private key is copied anywhere, it is as good as publicly available on the internet, allowing anyone to see the traffic going back and forth on your site. In this book, I *only* use private keys for self-signed certificates used in debugging. It is conceivable that similar debugging might need to take place on production, but every other possible avenue of exploration should be used before that happens.

If you give Wireshark the private key (see Appendix B, *Using Wireshark to Sniff SSL Packets*), then it is capable of decrypting the packets *if* an RSA key exchange is used. SSL connections are not complete after the server sends its certificate and the client validates it is with a certificate authority.

After trust is established between the browser and server, the two use that trust to build a session-specific set of keys that will be used to encrypt communication in both directions. The exchange of those keys itself needs to be encrypted (and signed by the server's certificate). It is not that complicated [5], but for our purposes, Wireshark needs to be able to follow along with that key exchange.

The "cipher suite" chosen to exchange keys needs to be one Wireshark is capable of deciphering. Even with the server's private key, certain cipher suites are not supported by Wireshark. There are a *lot* of cipher suites. Many have been obsoleted over the years [6], but there are still many supported by modern browsers and servers.

The cipher suite used is the strongest one supported by both client and server. Unless you are developing a web browser (or SPDY client), you generally have more control over the web server. This means that it is easier to configure the server to support a cipher suite that most clients will support and that Wireshark can follow along with. The "DES-CBC3-SHA" cipher suite is a good bet for both.

In Apache, this is configured with:

```
SSLCipherSuite DES-CBC3-SHA
```

In web frameworks, this is done somewhere in code. Fortuitously, node.js explicitly uses this cipher suite, making our lives easier. If you are using something different, the cipher suite may be buried quite deep in the code. It is worth getting familiar with whatever framework you may be using in case you need to examine packets someday.

# 8.3. Additional SSL Modifications

If there is a weak link in SPDY's quest for speed, it is SSL. Not necessarily the ongoing compression after a connection has been established—that is relatively quick. Rather the

[5] Wikipedia has a nice description http://en.wikipedia.org/wiki/Transport_Layer_Security#Security
[6] "Obsolete"/"old" cipher suites are easily crackable

initial connection incurs two round trips to say "hello" (exchange identification) and to establish a secure channel:

```
Client Hello
Server Hello, Certificate, Server Hello Done
Client Key Exchange, Change Cipher Spec, Encrypted Handshake Message
Encrypted Handshake Message, Change Cipher Spec, Encrypted Handshake Message
Application Data
Application Data
Application Data
```

Only after completing those two round trips is application data able to flow. This is a problem even with HTTPS, so it is not SPDY specific. Regardless, it would be ideal to keep this time down. Toward that end, Google has submitted at least two Drafts to the IETF to extend SSL for speed.

The first of the two proposals is known as "false start" [7]. Rather than wait for a second round trip to begin choosing cipher suites to be used, false start moves it into the server hello message. If the client agrees to the cipher suite, the server can begin sending application data immediately. This eliminates one network trip, which can be significant depending on the latency in a connection.

There are a limited number of cases in which this is possible and even then both client and server must be capable of performing false start. If anything goes wrong, both client and server have to go back to the starting line and do things the old fashioned way. Fortunately, HTTP and SPDY are excellent candidates for false start. Hopefully we will begin to see browsers and application servers support this soon.

The second IETF Draft, known as "snap start" [8], proposes to alter SSL negotiation even further. It suggests to put enough information in the client hello message so that the SSL connection can be established immediately after the server response. This would eliminate an entire round-trip, which would be a significant savings. The conditions under which this is possible are even more limited than with false start. Even so, it is intentionally crafted to work with most web situations.

Neither of these drafts has been implemented in the wild. As SSL and SPDY become more prevalent, something along these lines will have to be adopted to keep the web fast.

[7] http://tools.ietf.org/html/draft-bmoeller-tls-falsestart-00
[8] http://tools.ietf.org/html/draft-agl-tls-snapstart-00

# 8.4. Conclusion

SSL is mostly unobtrusive in SPDY. It makes SPDY secure by default. It is also responsible for advertising that the server is capable of speaking SPDY via NPN. On the off-chance that you should ever need to examine individual SPDY packets, it is possible.

But, for the most part, SSL in SPDY is the same SSL that you have always used. Certificates are obtained the same way that HTTPS certificates are obtained. It works under the covers the same way that SSL has always worked. But it lets us do some really cool things with SPDY on top of it!

In the next chapter, we will revisit application coding and see what some of those cool things might be.

# Chapter 9. SPDY Alternatives

SPDY is not the only player in the web page load time game. It turns out to be the only player that focuses only on web page load times. Ultimately this and its adherence to supporting existing HTTP infrastructures are what make SPDY the most viable of the alternatives.

Still there are others. Some of them can be implemented right away so let’s take a look.

## 9.1. CWND Sorting

Recent versions of the Firefox browser have introduced an optimization that is so obvious and simple that it is a wonder no one thought of it before. Firefox now sorts its tubes based on the congestion window of each. This ensures that the next connection always goes out over a relatively warm tube.

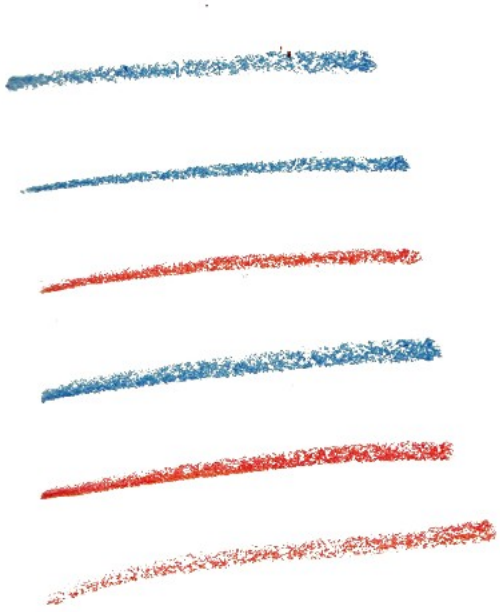
Recall from earlier discussion that a TCP/IP connection starts off "cold". In this context, "cold" refers to how much data is transferred before the connections pauses, awaiting acknowledgment from the other side that the data was received.

Also recall that all modern browsers open up 6 separate connections to a sub-domain for transferring data. These six HTTP tubes are no exception to the slow TCP/IP start. At first they may only be capable of transferring a few hundred bytes of data before pausing. Depending on the round trip time of the connection, those pauses can take anywhere from 30-100ms before the next batch of data is sent out.

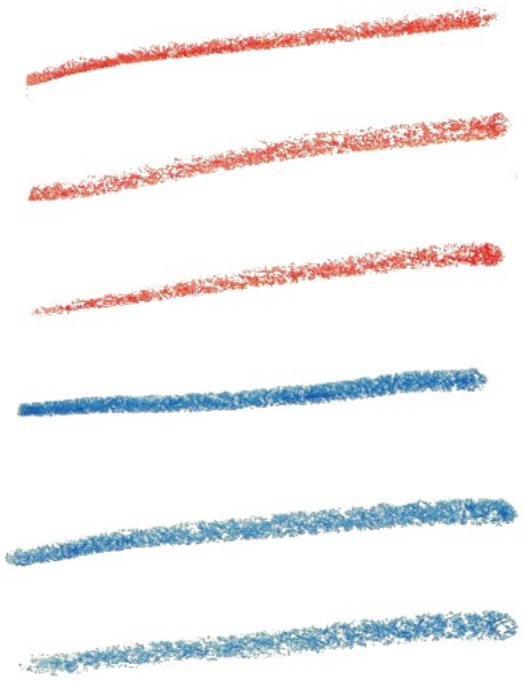
The next batch of data is generally sent back in a larger chunk (depending on connection info sent back by the other side). Eventually, the chunks of data sent before pausing, called the congestion window (CWND), grow to an optimal size for the bandwidth between client and server. At this point, the interweb tube is said to be "warm".

All interweb tubes are created equal, but they do not warm up equal. A tube that transfers a couple of small images does not facilitate enough round trips to warm up. A tube that transfers a large image or Javascript library, on the other hand, might be positively on fire.

When it comes to choosing the next tube to use, browsers are profoundly dumb—they simply choose the first one that went idle. Of course, this tends to be the coldest tubes since they did so little work.



The innovation introduced by Firefox was to simply change the ordering of the tubes. Rather than put the first idle connection at the top of the list, Firefox puts the warmest tubes at the top of the list.



## 9.1.1. Drawbacks

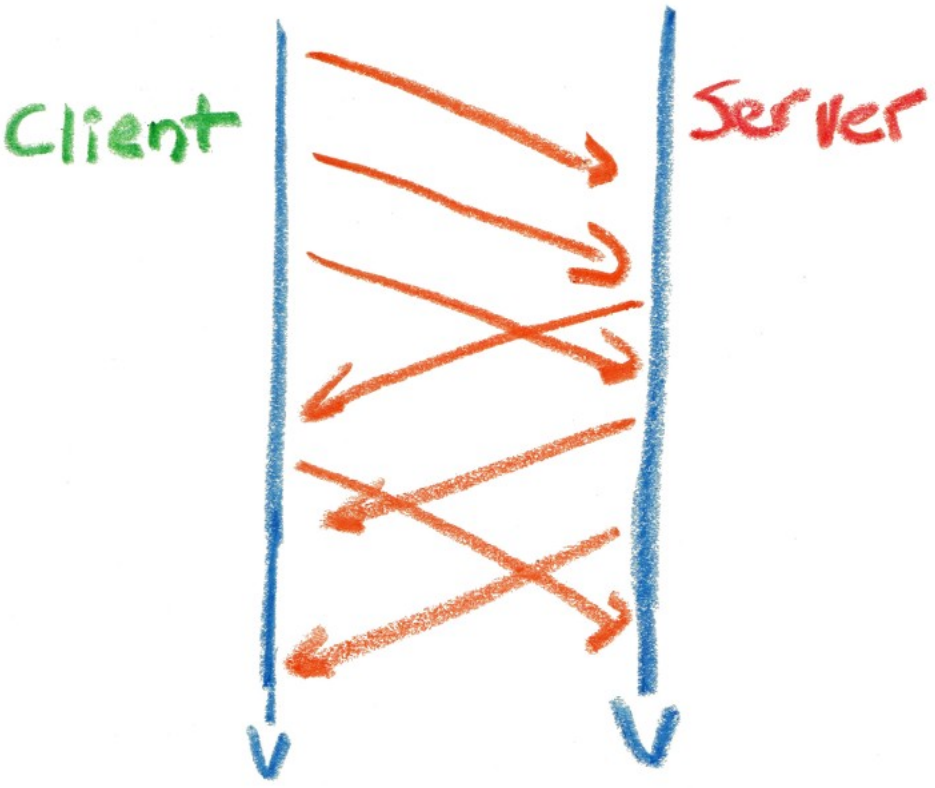
CWND sorting has no effect on the first connections made by browsers. How could it? All tubes are created equal. Subsequent connections to the server will certainly be faster that the first connection. Subsequent page loads will be faster—especially if most of the data is cached on the first requests.

But ultimately it does not help to multiplex requests. Browsers are still only able to request six resources at a time. Everything else pauses while the browser waits on those resources.

CWND sorting is a useful optimization, but it is nowhere near the same class as SPDY.

# 9.2. HTTP Pipelining

As impressive as HTTP/1.1 is, it is even more so, considering that the original authors included the concept of pipelining HTTP requests in the spec. Pipelining requests in a single TCP/IP stream eliminate much of the overhead involved in modern HTTP requests:




The browser does not have to wait for the first response to come back before requesting the second and third resources. Over the course of a single page load, that adds up to significant savings (which is why SPDY uses it).

### 9.2.1. Drawbacks

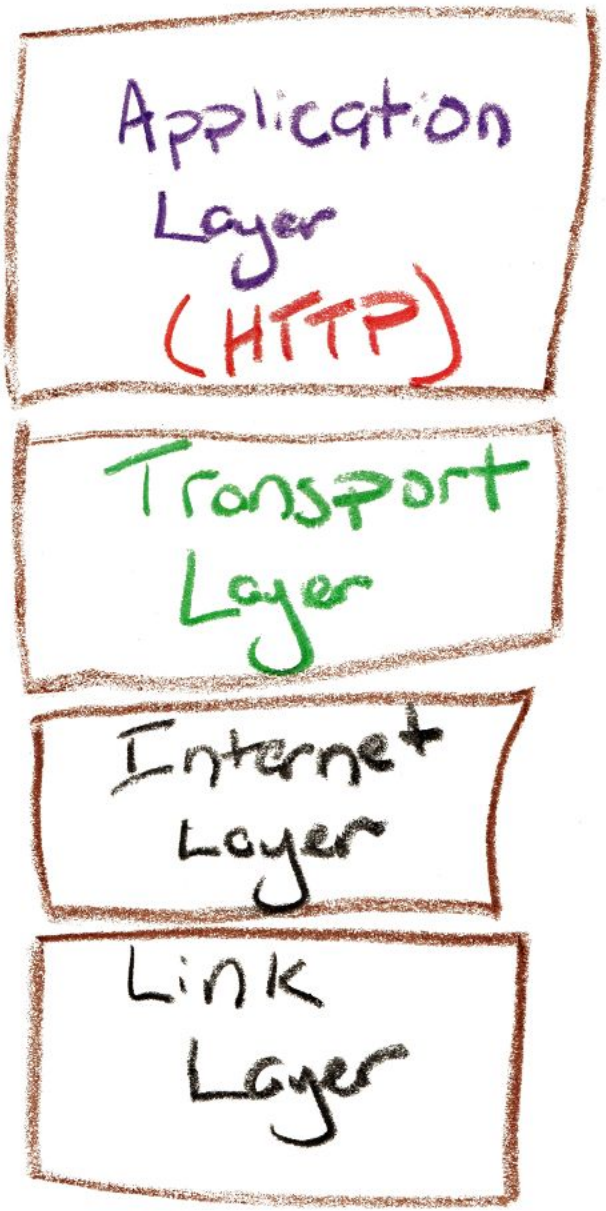
Pipelining is not multiplexing, which *is* part of SPDY. Multiplexing allows the responses to interleave. Without it, the first response blocks subsequent responses until it is complete. Thus, computationally long resources can dramatically slow pipelined pages.

Pipelining is also notoriously hard to implement in HTTP (otherwise all browser vendors would support it). The primary reason for the difficulty is the ability to determine whether or not the server is capable of handling pipelined requests. Opera and Firefox both support pipelining, but only in circumstances in which there is a high degree of confidence that it will work. Firefox does not even attempt pipelining until the second page request on a site, making it of no benefit to initial page load speeds.

## 9.3. Networking Fixes

There are a couple of network layer solutions that are mentioned quite a bit when discussing SPDY. Two brought up by SPDY haters are BEEP and SCTP. I will talk about each briefly. Bottom line: neither are designed from the ground up with page load time in mind.

SPDY, like HTTP, lives at the top of the networking stack:




It relies on TCP/IP to transport its packets and lower level stuff to actually move those packets on the wire.

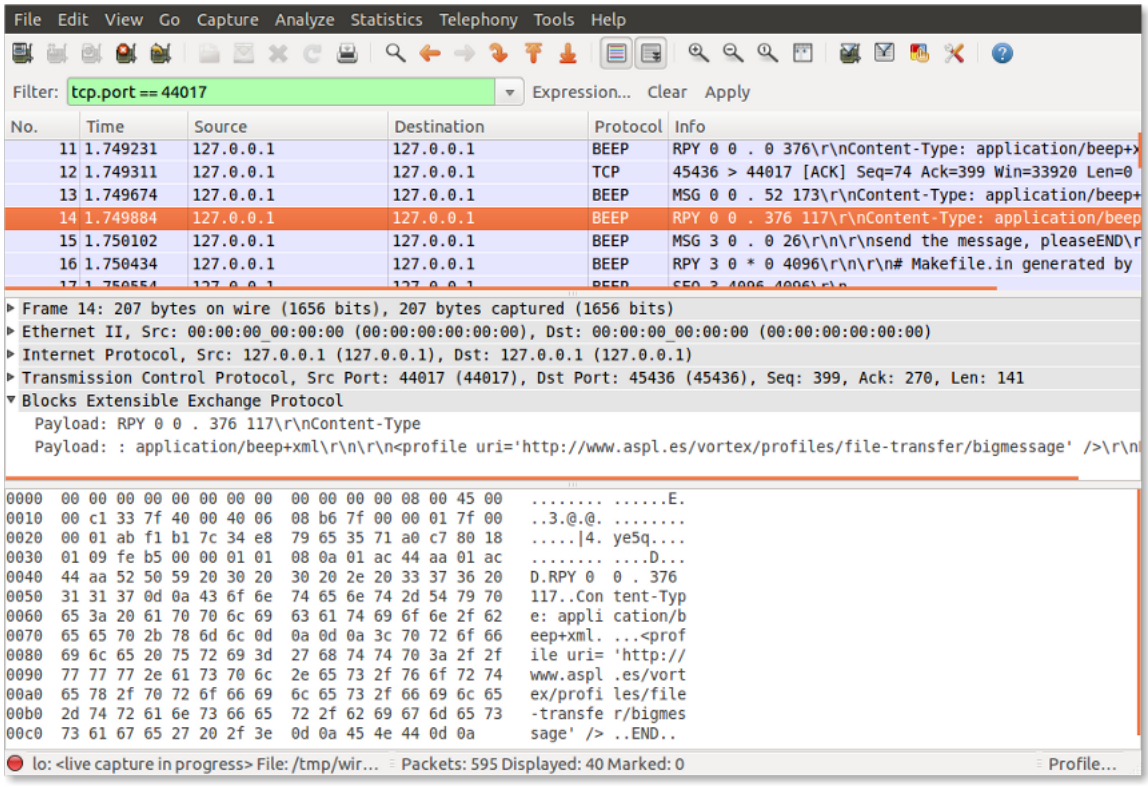
In contrast, these networking fixes start to move down the networking stack. This gives them some very nice features, though they are of limited benefit to application level concerns like page load times.

## 9.3.1. BEEP

BEEP (Block Extensible Exchange Protocol) is an interesting protocol primarily intended for building other protocols. In much the way that SPDY does, BEEP incorporates the best practices of the networking industry for the past 20 years.

With BEEP, you get multiplexing built-in. Also baked-in from the start are nice security features. It is built on top of TCP, so major changes are not required to network infrastructures.

The most obvious drawback to BEEP is that the packet headers are plain text:



As we saw in Chapter 7, *Control Frames in Depth*, one of the big wins in SPDY is the ability to compress HTTP headers. Even if this could be done quickly, BEEP would require changes to support header compression.

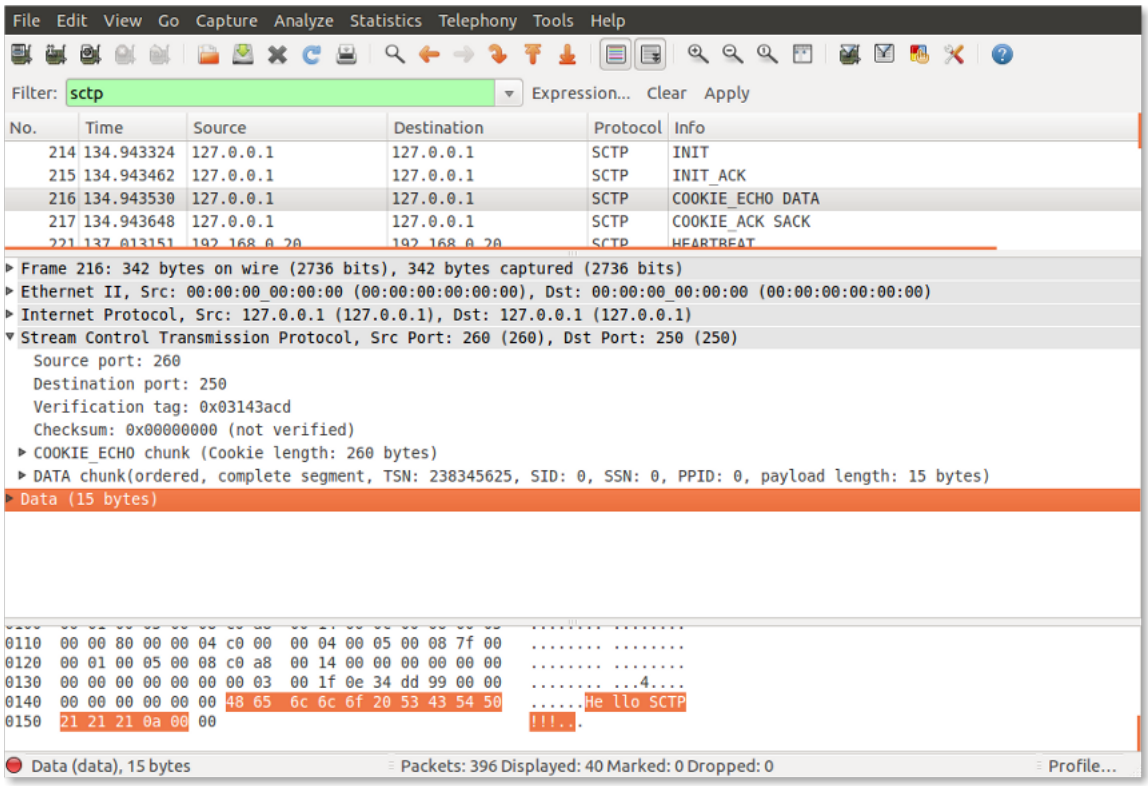
Ultimately, the next generation of HTTP should be built from the ground up with page load time in mind (even if not the primary motivation). Attempting to tack that onto a more general framework is not likely to produce the best results.

Which leads us to…

## 9.3.2. SCTP

Any time articles on SPDY are posted, it seems to bring out the inevitable complaint that Google are imbeciles for ignoring SCTP (Stream Control Transmission Protocol). I have to confess that I just don't see it.

Like BEEP, it is well supported by Wireshark (something that SPDY cannot claim):




SCTP also has some nice security features. It is built at the transport layer, so it is more a competitor to TCP than SPDY in that respect. As seen in the Wireshark window, SCTP connections are initiated with 4 packets instead of 3 like in TCP/IP. This eliminates the dreaded SYN flooding attack, for which there is no real defense in TCP/IP, just mitigation techniques.

A SYN flood takes place when an attack sends a large number of SYN packets, which signal a connection start in TCP/IP. The problem with this is that servers need to allocate resources at this point—even if the client has no intention of completing the connection. Too many of these packets and the server is soon overwhelmed.

In SCTP, the client is required to establish a connection on its end before the server allocates resources for the connection on its side. This makes connection flooding prohibitively hard for an attacker since the client would require far more resources.

Regardless of some of the nice features in SCTP, it was not built from the ground up with page load times in mind. This is not a short-coming of SCTP, simply a statement of fact. SCTP is an alternative to TCP/IP *not* HTTP.

One of the drawbacks to being a TCP/IP competitor is that the internet infrastructure is built on TCP/IP. The entire internet is not going to switch overnight to a new protocol. This is not the early days of the web where everyone could be convinced to switch to Winsock [1].

Even if this were possible, the internet is built on top of Network Address Translation (NAT), which is not possible with SCTP (or anything other than TCP and UDP).

SCTP does come with the promise of being able to run over UDP [2]. But this support is theoretical only at this point.

In the end, SCTP does have some nice overlap with SPDY's goals, but not enough that can overcome the gaps.

# 9.4. Conclusion

Of the alternatives that are out there, tweaks to existing HTTP implementations hold the most promise. Even though HTTP Pipelining support is quite spotty even in the browsers that claim some support, it does hold some of the promise of SPDY's ability to make more efficient use of interweb tubes. If it is possible to tune Pipelining, browser vendors may realize something close to the performance gains of SPDY. That, without needing a radical new protocol, would be a win. Throw in tweaks like CWND sorting and, who knows, maybe they will get there.

Networking solutions that are lower on the stack than HTTP, however, seem dead on arrival as competitors to SPDY. They have interesting features—some that would be very nice to have in SPDY—but they lack the specialization that is needed for the next generation of web application. They lack the ability to significantly drive page load times.

[1] In the early days of Windows, it did not support TCP/IP. To do anything with TCP/IP, including browse the web, users needed to install special drivers. It was a different time with a far greater percentage of technical users that allowed this switch to occur.
[2] User Datagram Protocol

# Chapter 10. The Future

SPDY is still very much in development. The spec continues to evolve and major features are still being removed (thankfully, no new ones are going in). So the most obvious aspect of SPDY's future is change.

That said, the overall approach and many details are fairly set. Certainly, there will be aspects of this book that will be out of date before the end of the year (hopefully not too many).

Before waxing poetic on the indomitable future of SPDY, let's consider some of the implications of what we have learned. Building on some of those assumptions, we can already see what the next generation SPDY apps might look like. We can revisit our SPDY server push and play some tricks that might get you thinking about some cool stuff coming up.

Then I wax poetic about the future.

## 10.1. Imagine a World

In mid-2011, Chrome held 20% of the market share [1]. Some analysts are predicting that its steep rise shows no signs of slowing and that, by mid-2012, Google's web browser will account for as much as 60% of the market. If that happens and one or more of its competitors adopts SPDY, it is easy to envision 80% or more of the market supporting SPDY.

At that point, it makes fiscal sense for website operators to give up on CDNs and other HTTP optimizations and switch entirely to SPDY for delivery of high-quality, fast web sites.

If all that happens (or even some of that happens), what kind of things might we see pop up?

One likely thing that we'll see is more manipulation of the stream prioritization. I have only mentioned it in passing in this book, primarily because the programming

[1] http://gs.statcounter.com/press/chrome-breaks-20-perc-globally-in-june

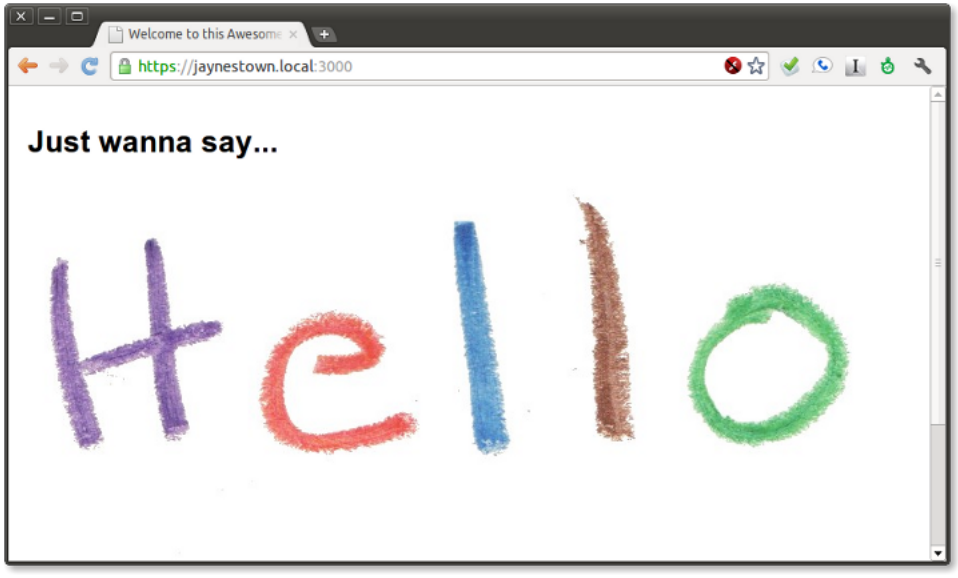
implementations available at this time do not support it. But certainly, if there is any more performance to be eked out of SPDY, that is going to be the plaything of performance experts.

I still believe that there is more to be done with SPDY server push.

# 10.2. Stupid Push Tricks

One of the many interesting aspects of SPDY push is its ability to defer the push until the server is ready to send it along. What this means in practice is that the server can jam out a bunch of static data to the browser almost immediately. After performing the necessary computation, the server can then send out dynamic information. The win for SPDY, of course, is that the wait for the computationally intensive data does not block any other data from flowing.

To see this in action, consider a static page with very little on it. Even static pages need a little help. In this case, the static page will also include CSS, a touch of Javascript, and several images (as in previous chapters, each letter in "Hello" is a separate image):



In `app.js`, configure the server be SPDY-ized and to perform push directly into the browser cache (as in Chapter 4, *SPDY Push*):

```
var express = require('express-spdy')
  , fs = require('fs')
  , createPushStream = require('spdy').createPushStream
  , host = "https://jaynestown.local:3000/";

var app = module.exports = express.createServer({
  key: fs.readFileSync(__dirname + '/keys/jaynestown.key'),
  cert: fs.readFileSync(__dirname + '/keys/jaynestown.crt'),
  ca: fs.readFileSync(__dirname + '/keys/jaynestown.csr'),
  NPNProtocols: ['spdy/2'],
  push: awesome_push
});
```

There is nothing in that configuration that we have not seen already. By now, you know that the interesting stuff is going to be in the `awesome_push` callback. Before looking at that, let's look at the (mostly static) site content.

As promised, the static page is comprised of very simple HTML:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Welcome to this Awesome Site!</title>
    <link rel="stylesheet" href="/stylesheets/style.css"/>
    <script src="/javascripts/jquery-1.6.2.min.js"></script>
  </head>
  <body>

    <h1>Just wanna say...</h1>

    <div id="hello">
      <img src="/images/00-h.jpg">
      <img src="/images/01-e.jpg">
      <img src="/images/02-l.jpg">
      <img src="/images/03-l.jpg">
      <img src="/images/04-o.jpg">
    </div>

  </body>
```

```
</html>
```

To make the page a bit more dynamic, let’s add the following "profile" section to the HTML:

```
<div id="profile" style="display:none">
  <p>
    Welcome back, <span id="name"></span>.
  </p>
  <p>
    Today is <span id="date"></span>.
  </p>
  <p>
    Your favorite color is: <span id="favorite_color"></span>.
  </p>
  <p>
    I worked really hard to come up with this.
    I think your favorite number might be
    <span id="random_number"></span>.
  </p>
</div>
```

Unless something else happens, this profile HTML will not display (due to `style="display:none"`). Even if it did display, it would be very boring since all of the `<span>` tags are empty. A little bit of Javascript can be used to populate those spans and to display a user profile:

```
<script src="profile.js"></script>
<script language="javascript">
  $(function() {
    if (profile.name) {
      $("#profile").show();
      $("#name").text(profile.name);
      $("#date").text(profile.date);
      $("#favorite_color").text(profile.favorite_color);
      $("#random_number").text(profile.random_number);
    }
  });
</script>
```

On the file system, profile.js will be empty:

```
$ cat public/profile.js
```

```
var profile = {};
```

An empty profile object will result in no change to the page display (thanks to the `if (profile.name)` conditional).

We know that we can push data into the browser cache with SPDY server push. In Chapter 4, *SPDY Push*, we only pushed static content from the file system. There is nothing preventing us, however, from pushing anything we like in its place—including dynamic JSON.

So, back to the express-spdy application. The `awesome_push` callback will contain:

```
function awesome_push(pusher) {
  // Only push in response to the first request
  if (pusher.streamID > 1) return;

  // Oh boy, this is going to take a while to compute...
  long_running_push(pusher);

  // Push resources that can be deferred until after the response
  pusher.pushLater([
    local_path_and_url("stylesheets/style.css"),
    local_path_and_url("javascripts/jquery-1.6.2.min.js"),
    local_path_and_url("images/00-h.jpg"),
    local_path_and_url("images/01-e.jpg"),
    local_path_and_url("images/02-l.jpg"),
    local_path_and_url("images/03-l.jpg"),
    local_path_and_url("images/04-o.jpg")
  ]);
}
```

First up, we start a long running push operation with a call to `long_running_push()`. This is node.js, so it will not prevent the rest of `awesome_push` from executing. Once `long_running_push` is doing its thing, we push the stylesheet, Javascript, and images directly into the browser cache. This is a "later" push in that the data will be sent after the original HTML response is complete.

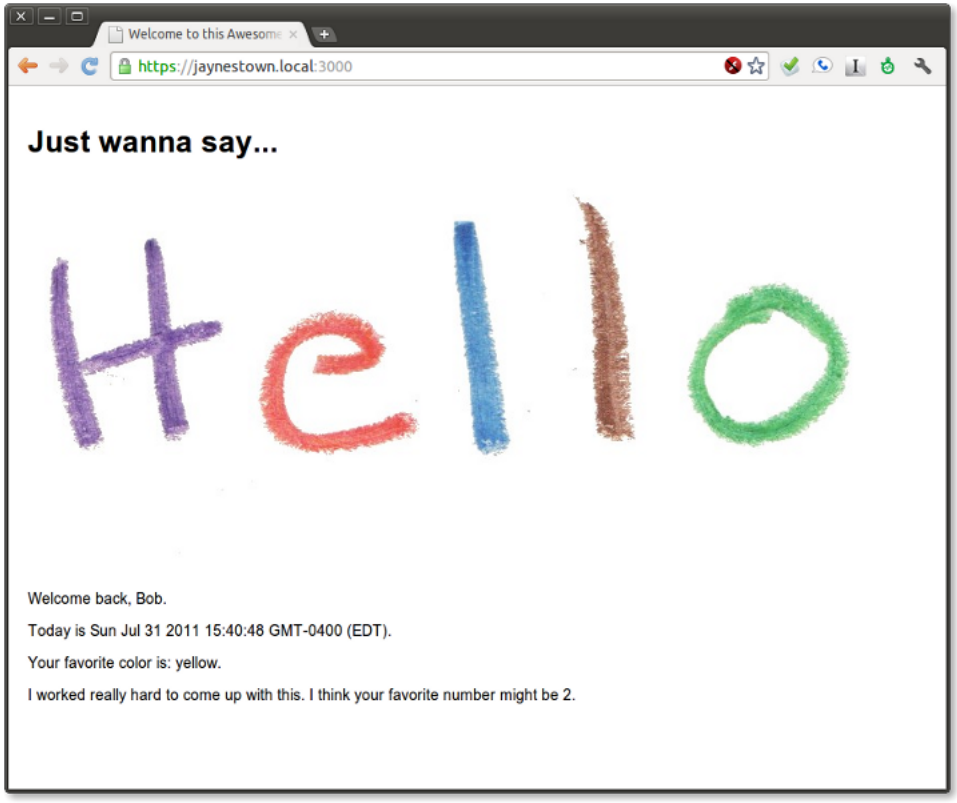
Finally, in the `long_running_push()` function, we push out the dynamic data once it is ready:

```
function long_running_push(pusher) {
```

```
  var url = host + "profile.js"
    , push_stream = createPushStream(pusher.cframe, pusher.c, url);

  // Send the push stream headers
  // Ew. Need to expose an API for this...
  push_stream._flushHead();
  push_stream._written = true;

  setTimeout(function () {
    // Write the push stream data
    push_stream.write(
      'var profile = {' +
      '  name: "Bob",' +
      '  favorite_color: "yellow",' +
      '  date: "' + (new Date).toString() + '",' +
      '  random_number: "' + Math.ceil(Math.random() * 10) + '"' +
      '}'
    );
    push_stream.end();
  }, 3*1000);
};
```

As you can see, the long running push is simply being simulated with a 3 second call to `setTimeout()`. Once 3 seconds have elapsed, the `push_stream` object will write its data. The "data", in this case, is the assignment of a Javascript object literal to the `profile` variable. Once pushed to the browser, our simple jQuery code will use the `profile` object to populate and display the page's profile section.

Loading up the app in Chrome, we see the same homepage—for 3 seconds. After 3 seconds have elapsed, and the computationally intensive `long_running_push` finally sends back its data, the page looks like:




Granted, it took a long time for our poor little server to calculate our favorite number, but that's not the point.

The point is that the user experience was excellent. The entire page rendered in under a second because the browser had to make only one request to get everything. There was no round trip to request the various Javascript, CSS, and image files. A single request resulted in everything being jammed into browser cache via SPDY server push.

Even more exciting is that dynamic data was injected in the browser cache and (this is important) it did so without blocking any requests. Think about that. In vanilla HTTP, a computationally intensive request can block all requests on an interweb tube. Since browsers only get 6 interweb tubes per site, a few of these types of requests and the page grinds to a halt. But not with SPDY.

With SPDY server push, nothing blocks. Everything flies across the wire when it is ready —no waiting on round trip times, blocking resources or anything. Just raw speed.

As Javascript frameworks grow in popularity, this kind of dynamic push will be vital to delivering quick applications instead of the blank pages that we see all too often in 2011.

# 10.3. Conclusion

In mid-2011, only 20% of the browser market supports SPDY. If you ignore SPDY and spend time and money optimizing for HTTP, you have a huge investment that reaches the entire market.

But what if Chrome reaches 40% of the market? 50%? At that point, a switch to SPDY would mean that you can achieve amazing page load speeds without expending tons of precious time or money doing everything in your power to optimize performance. Instead, you can focus on improving your product, which may include adding features only possible with a fast protocol like SPDY.

In that case, you can argue that you need almost *zero* time and money spent on optimization, but still deliver a blazingly fast experience for your users. And the other 50%-60% of the market? You can say that they don't care about performance. They are content with the performance given them by HTTP/1.1 era technology.

So in the end, you give the users that care the most a SPDY experience. The rest receive an experience that is good enough. And you spend no money other than your hosting fees. Sure you might lose a few customers from those other browsers. But don't tell me you care about them (unless you are already working your butt off to HTTP optimize).

At what point does it become cost-effective to ignore browsers that do not support SPDY?

I think the answer to that depends on you and your product. But, if I am building a website in early 2012, I am not bothering with HTTP optimization. I will deliver incredible speed, reliability and security to my power users with SPDY. My other users will have to be content with my awesome product.

# Appendix A. Installing Express-SPDY

*OK kid, this is where it gets complicated* — Amy Pond

These instructions will install edge-openssl, edge-node.js and express-spdy in your home directory. This will **not** overwrite or affect any system libraries.

# A.1. Dependencies

Dependencies are described for a vanilla Debian 64-bit system, but should work equally well for Ubuntu. It should be relatively easy to extrapolate dependencies for OSX.

If not already installed, you need:

- C and C++ compilers for openssl and node.js
- Zlibh for SPDY compression
- Git to obtain the latest node.js source
- Python to configure node.js
- Curl for installing the Node Package Manager (NPM)

This can be accomplished on Debian / Ubuntu with:

```
sudo apt-get install build-essential zlib1g-dev git-core python curl
```

# A.2. Edge openssl

SPDY requires Next Protocol Negotiation (NPN) extensions to work properly. This is only available in edge openssl. This can be obtained from the openssl CVS repository, but that would require installing CVS. It is easier to access the [openssl FTP server] (ftp://ftp.openssl.org/snapshot/). Look for tar.gz files that begin with `openssl-SNAP-`. Following that prefix is a datestamp (the tarball from June 22 was `openssl-`

`SNAP-20110622.tar.gz`). Unfortunately those tarballs are only available in a rolling 4 day window so permanent links are not possible.

Once you have identified the correct openssl SNAP tarball, download and un-tar it in `$HOME/src`:

```
mkdir -p $HOME/src
cd $HOME/src
wget ftp://ftp.openssl.org/snapshot/openssl-SNAP-<datestamp>.tar.gz
tar zxf openssl-SNAP-<datestamp>.tar.gz
cd openssl-SNAP-<datestamp>
```

Next configure openssl for your platform. For 32-bit linux, use:

```
./Configure shared \
  --prefix=$HOME/local \
  no-idea no-mdc2 no-rc5 zlib \
  enable-tlsext \
  linux-elf
```

If you are using 64-bit linux, replace `linux-elf` with `linux-x86_64`. If unsure which platform to use, run `./Configure` without any options to get a complete list.

After configuring openssl, build and install it:

```
make depend
make
make install
```

# A.3. Node.js

SPDY support is only available in node.js 0.5 and higher. To install, first obtain the source code from the node.js homepage [1].

At the time of this writing, the most recent version of node.js was 0.5.2 [2]. To install the source code in `$HOME/src`:

[1] http://nodejs.org/#download

[2] Node.js is under extremely rapid development, so please forgive the version number, which was, no doubt, outdated 2 days after this book was released. Node.js: making any book obsolete within a month

```
mkdir -p $HOME/src
cd $HOME/src/
wget http://nodejs.org/dist/v0.5.2/node-v0.5.2.tar.gz
tar zxf node-v0.5.2.tar.gz
cd node
```

Configure node to use edge-openssl and to install locally:

```
./configure \
  --openssl-includes=$HOME/local/include \
  --openssl-libpath=$HOME/local/lib \
  --prefix=$HOME/local/node-v0.5.2

make
make install
```

Be sure to change the version number in the `--prefix` argument accordingly.

# A.4. Configure Bash to Use Edge openssl and node.js

Add the following to $HOME/.bashrc:

```
# For locally installed binaries
export LD_LIBRARY_PATH=$HOME/local/lib
PATH=$HOME/local/bin:$PATH
PKG_CONFIG_PATH=$HOME/local/lib/pkgconfig
CPATH=$HOME/local/include
export MANPATH=$HOME/local/share/man:/usr/share/man

# For node.js work. For more info, see:
# http://blog.nodejs.org/2011/04/04/development-environment/
for i in $HOME/local/*; do
  [ -d $i/bin ] && PATH="${i}/bin:${PATH}"
  [ -d $i/sbin ] && PATH="${i}/sbin:${PATH}"
  [ -d $i/include ] && CPATH="${i}/include:${CPATH}"
  [ -d $i/lib ] && LD_LIBRARY_PATH="${i}/lib:${LD_LIBRARY_PATH}"
  [ -d $i/lib/pkgconfig ] && PKG_CONFIG_PATH="${i}/lib/pkgconfig:${PKG_CONFIG_P
  [ -d $i/share/man ] && MANPATH="${i}/share/man:${MANPATH}"
done
```

Logout and log back in to ensure that your changes are applied correctly.

# A.5. Install NPM and Express.js

The Node Package Manager (NPM) is installed via a bash script delivered by `curl`:

```
curl http://npmjs.org/install.sh | sh
```

After that, installation of node packages is straight forward. First, install the express.js package globally so that the `express` command will be in your path:

```
npm install -g express
```

# A.6. Create a Sample Express.js App to SPDY-ize

Next, we create an actual express-spdy application:

```
cd
mkdir -p $HOME/repos
cd $HOME/repos
express example-spdy
cd example-spdy
npm install express-spdy jade
```

Copy SSL keys from node-spdy:

```
mkdir keys
cp \
  node_modules/express-spdy/node_modules/spdy/keys/spdy* \
  keys/
```

Edit `app.js`. Replace the following at the top of the generated code:

```
var express = require('express');

var app = module.exports = express.createServer();
```

Replacing it with:

```
var express = require('express-spdy')
  , fs = require('fs');

var app = module.exports = express.createServer({
  key: fs.readFileSync(__dirname + '/keys/spdy-key.pem'),
  cert: fs.readFileSync(__dirname + '/keys/spdy-cert.pem'),
  ca: fs.readFileSync(__dirname + '/keys/spdy-csr.pem'),
  NPNProtocols: ['spdy/2']
});
```

# A.7. Run the Sample App

```
node app.js
```

If all has worked, you should see:

```
Express server listening on port 3000 in development mode
```

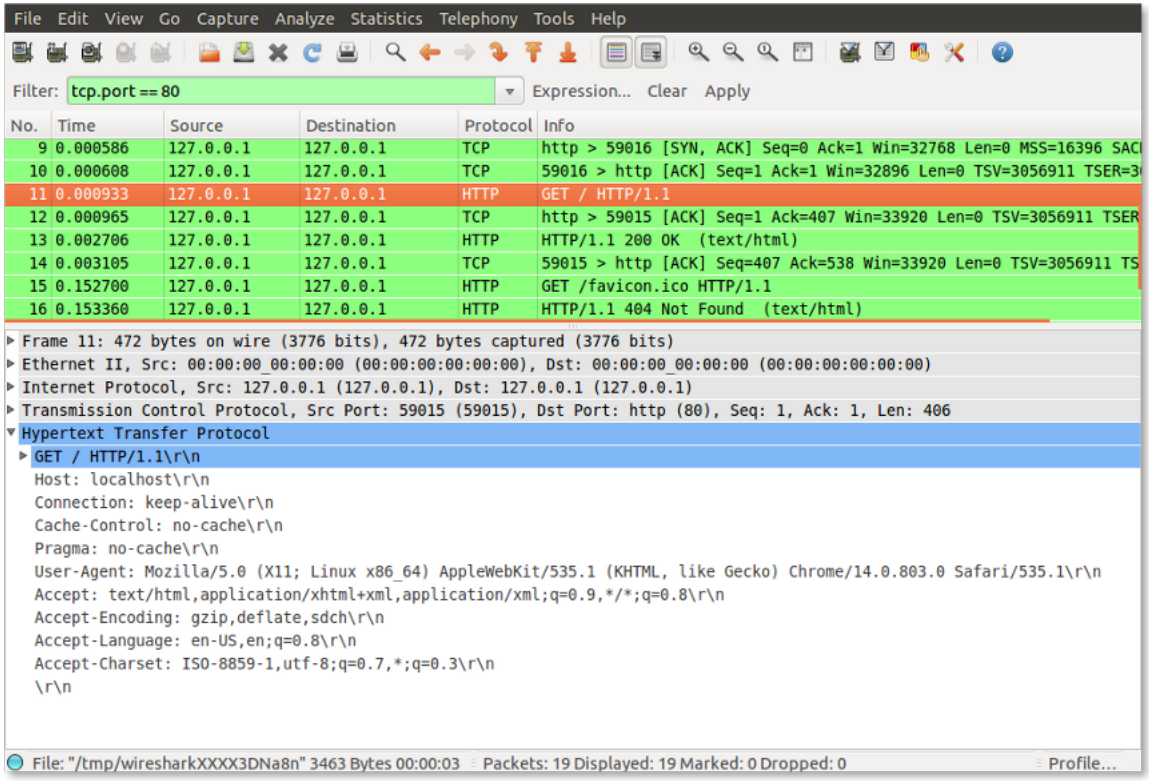
And you should be able to access the SPDY-ized site at: https://localhost:3000. You will have to proceed past a certificate warning. Such warnings can be eliminated by following the steps described in Section 8.2.2, “Being Your Own CA”.

To verify that you are seeing a SPDY session and not just a plain-old HTTP session, checkout the SPDY tab in Chrome’s `about:net-internals`.

# Appendix B. Using Wireshark to Sniff SSL Packets

Wireshark [1] is a fantastic network analysis tool. Actually, it is more of a network packet analyzer with some traffic analysis tacked on. That is not meant as a knock. I love me a good packet analysis. As an old `tcpdump` command-line aficionado, it took something really special to get me to switch.

Wireshark has panes of increasing detail. The top pane contains a list of all packets seen on the wire. In this case, traffic is being filtered such that only port 80 shows so we see all of the traffic associated with a web request.




[1] http://www.wireshark.org/

Clicking on a single packet in the top pane yields a packet information in the pane below. There is oodles of information to be had here. Everything from the bottom of the networking stack (Ethernet frame data) all the way up to the application layer. Wireshark even has intelligent parsing ability and is able to recognize various protocol (at all levels of the network stack) and report valuable information back to the user.

SPDY is secure by default because it runs over SSL (Secure Socket Layer). Unfortunately, the very nature of SSL makes it difficult to see the actual data packets that are being sent back and forth between browser and web server.

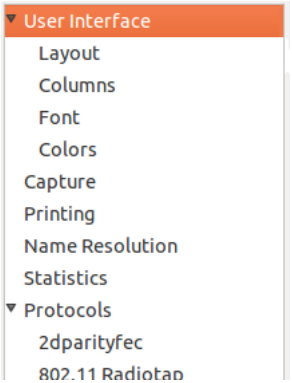
Chrome comes with the SPDY tab in `about:net-internals` which is quite useful for troubleshooting SPDY conversations. But sometimes there is no substitute for actually seeing the packets.

### Note

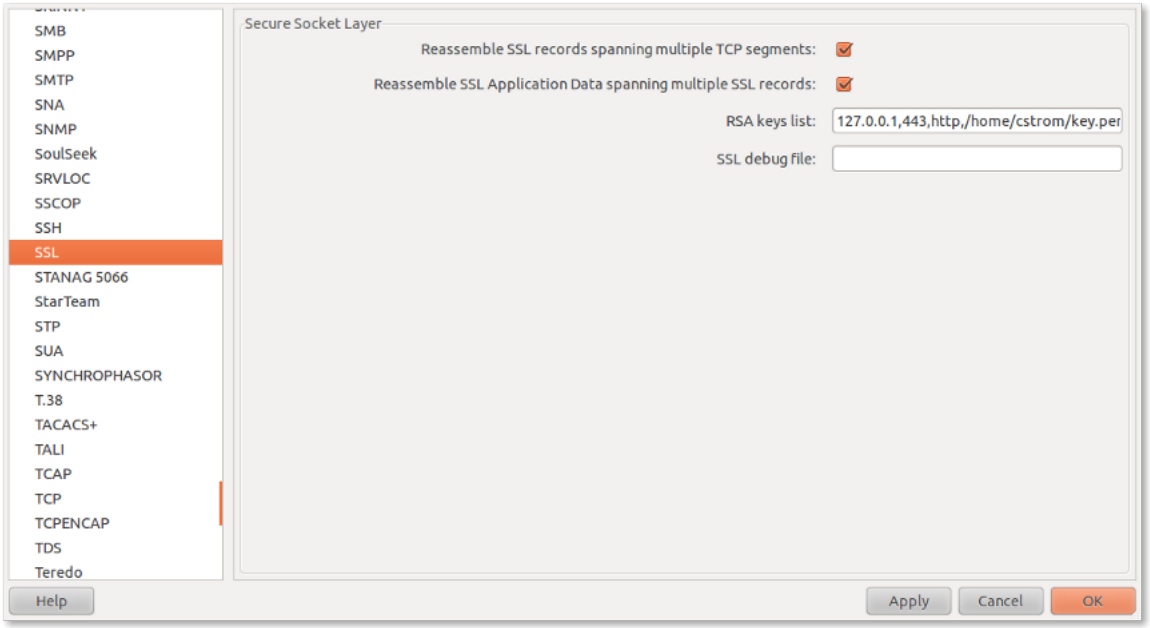
If you are compiling Wireshark for SSL support, then you will need `gnuTLS` installed first. With that, Wireshark needs to be compiled with the `--with-gnutls` command line option. On Ubuntu, Wireshark is packaged with `gnuTLS`. On other operating systems, you may need to manually install it.

Happily, Wireshark is capable of decrypting SSL conversations—*if* it can read the private key that is being used for the conversation. Under normal circumstances, private SSL keys should be guarded extremely well. When developing SPDY apps, it is good practice to generate your own certificates (see Section 8.2, “Creating SSL Certificates”). Since these certificates were created locally, it is easy to give Wireshark access.

To decrypt SSL conversations, Wireshark needs to be able to associate a private key with a port, network address and protocol (always http for HTTP/SPDY). This is configured by Edit # Preferences. In the Preferences dialog, the sidebar contains a list of Protocols:

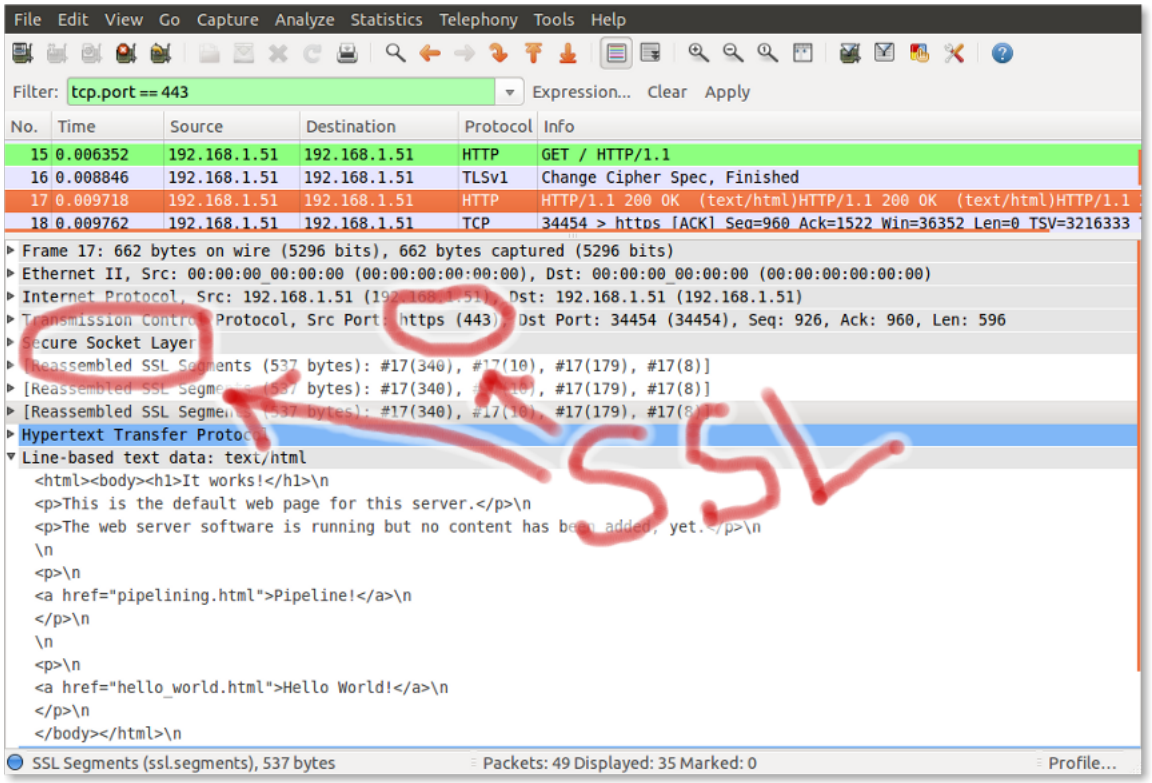



The protocols are in alphabetical order, so SSL is way down toward the bottom of the impressive list:



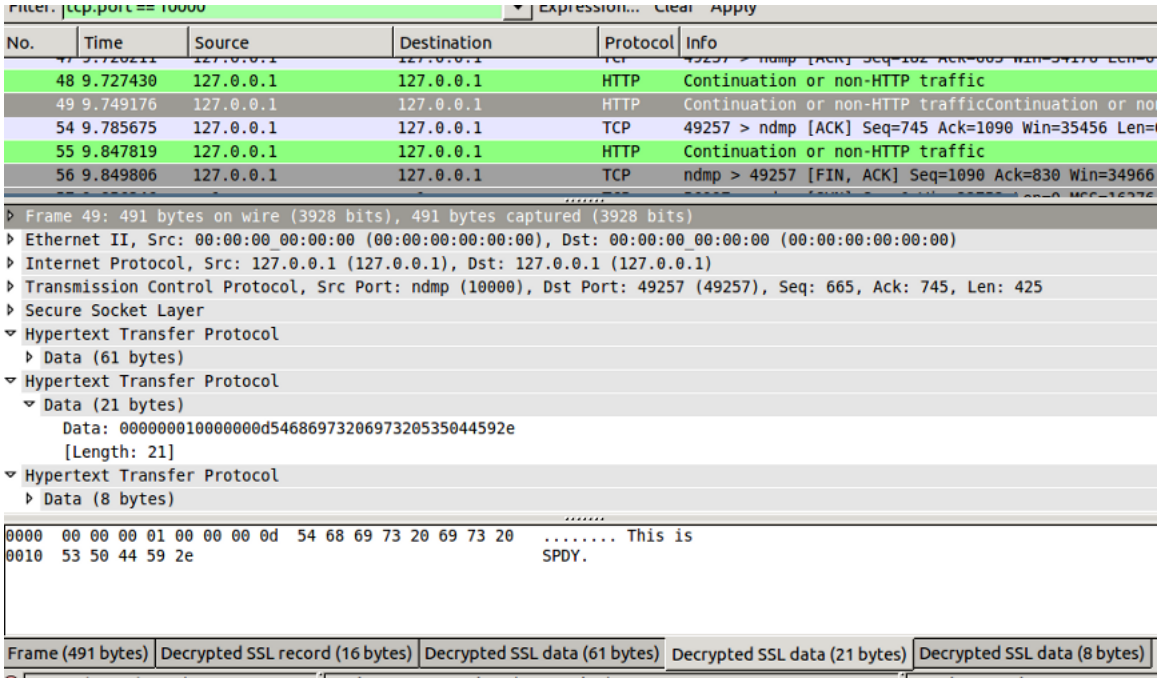

Network address, port, protocol and path to the private key are set in the "RSA keys list" as comma separated values (e.g. `127.0.0.1,3000,http,/home/cstrom/jaynestown.key`). Multiple combinations are separated by semi-colons: `127.0.0.1,3000,http,/home/cstrom/jaynestown.key; 192.168.1.100,443,http,/home/cstrom/spdy.key`.

Once you click Apply, Wireshark should be able to read and decrypt matching SSL conversations:




The HTTP response looks just like a normal conversation. If you are not paying attention (and your screen lacks badly drawn red arrows), it can be easy to miss that this is an encrypted conversation.

Unfortunately, at the time of this writing, Wireshark cannot understand SPDY. It can show you the contents of the packets, which is generally good enough:




## Important

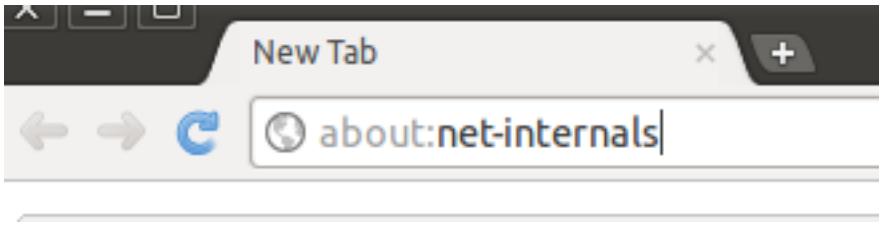
SSL client and servers negotiate the connection using different "cipher suites". Wireshark can only decipher RSA cipher suites. Node.js use such a cipher suite by default, but other implementation may not. More information on this can be found in Section 8.2.3, “Certificates That Play Nice with Wireshark”.

# Appendix C. Google Chrome

As the first and (as of this writing) only web browser capable of SPDY conversations, Google Chrome is an invaluable tool when exploring SPDY.

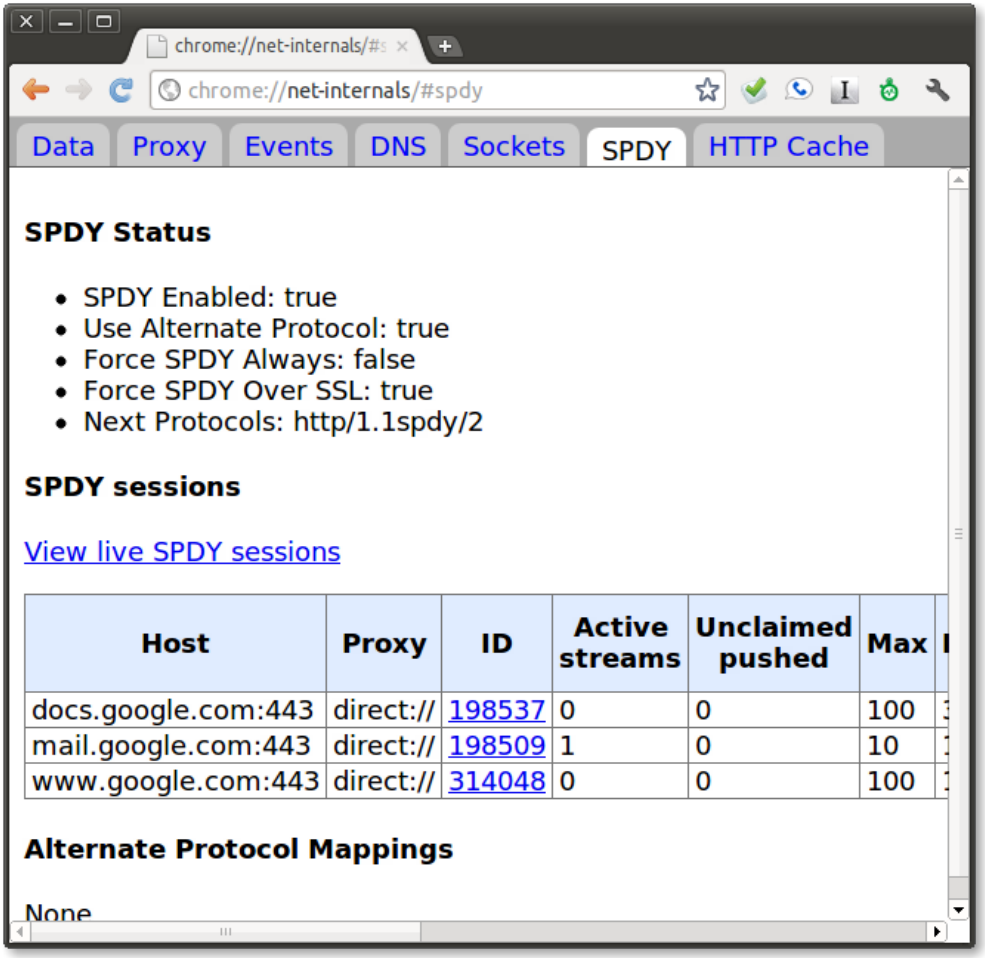
Most of the examples in this book were written back in the Chrome 14/15 time period.

The biggest asset to the proficient SPDY-ist is the SPDY tab in Chrome's about:net-internals. To access it, enter about:net-internals as the URL in Chrome's address bar:




Chrome's internal networking settings and information has numerous tabs. When working with SPDY, we will mostly concern ourselves with the SPDY and Events tabs.

Clicking on the SPDY tab will list any active SPDY sessions. Until SPDY adoption becomes more widespread, these will mostly be Google sites.




### Important

Before debugging or viewing a SPDY session, always open about:net-internals. Otherwise Chrome will not log the actual contents of SPDY sessions for you.

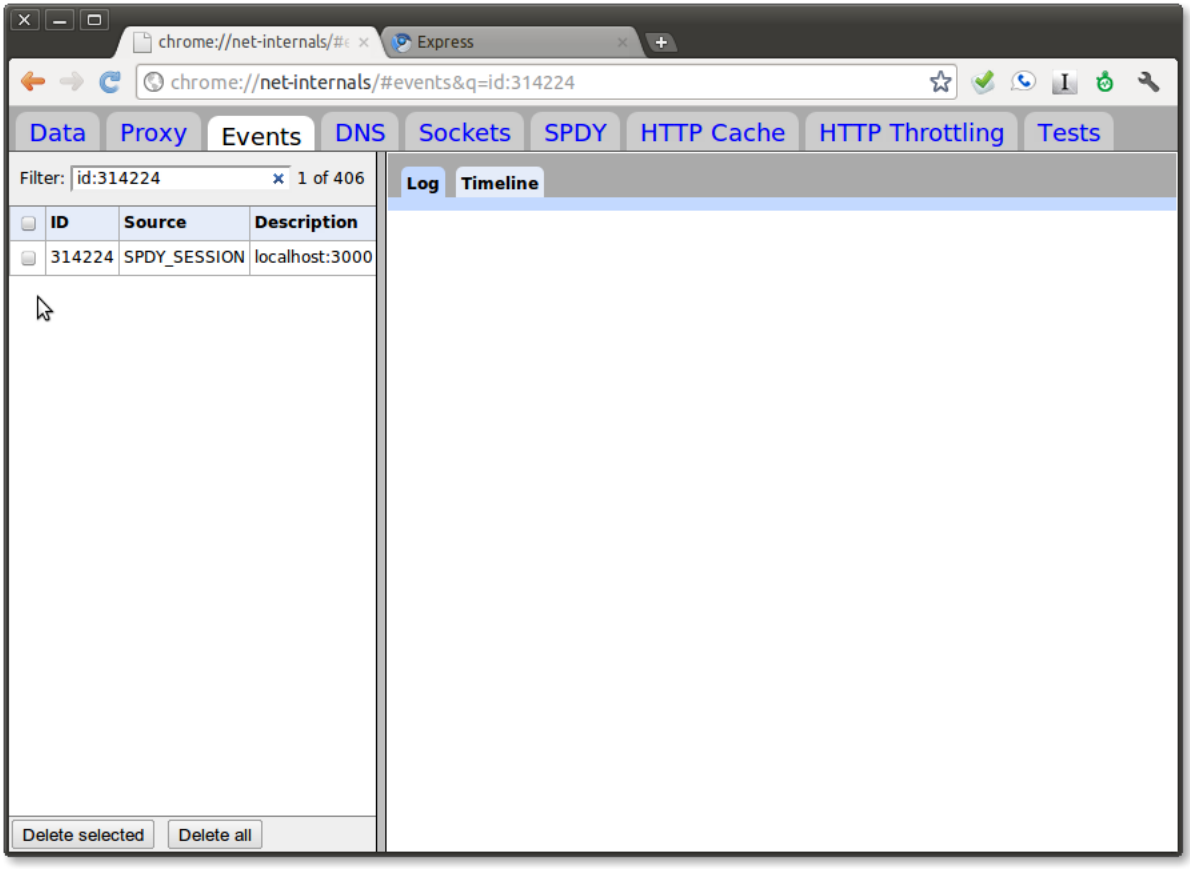
After initiating a SPDY session, it will appear in the list of active SPDY sessions:

**SPDY sessions**

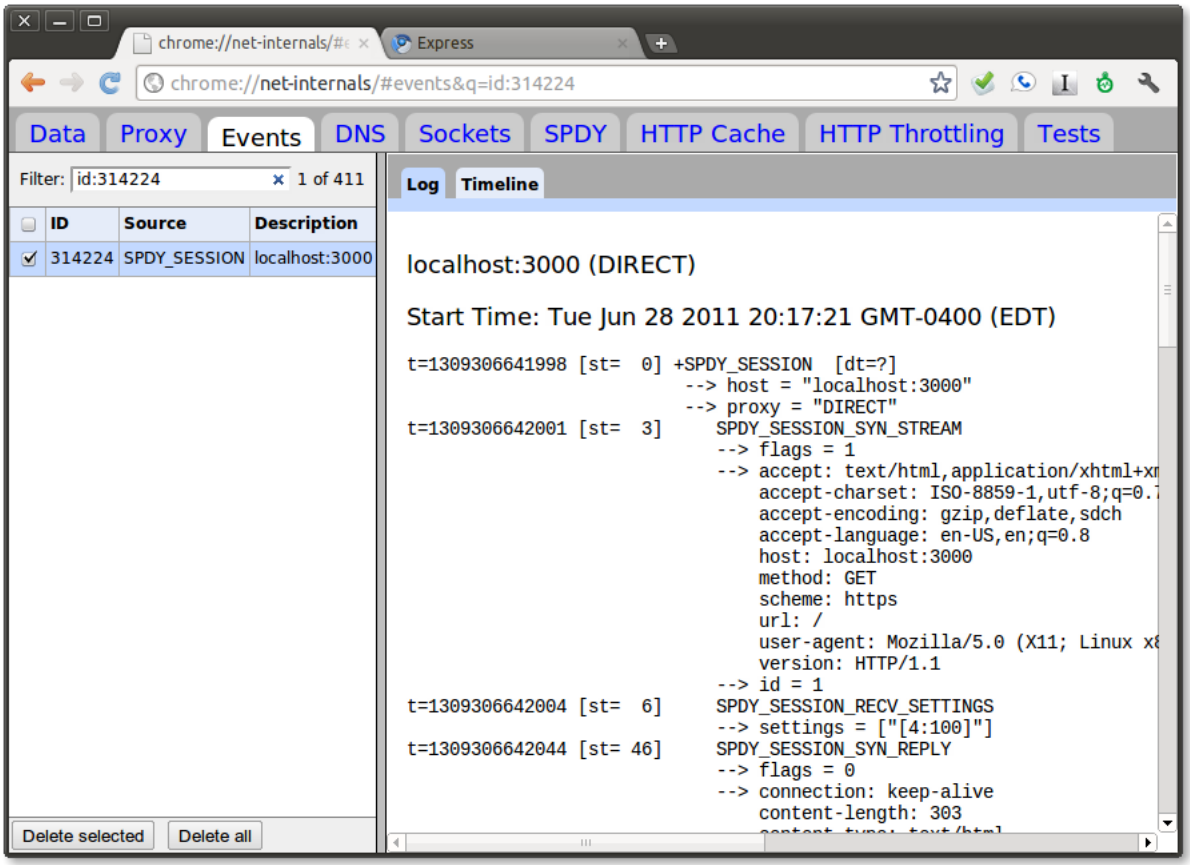
View live SPDY sessions

| Host | Proxy | ID | s |
|---|---|---|---|
| docs.google.com:443 | direct:// | 198537 | ( |
| mail.google.com:443 | direct:// | 198509 | 1 |
| localhost:3000 | direct:// | 314224 | ( |

Clicking on the new session will take you to the Events tab with all events save the new SPDY session filtered out.



Selecting the SPDY session will give you a very nice overview of what has occurred in your SPDY session. It will also give you live updates as new data passes over the same SPDY session.




Another useful tool is the Speed Tracer Chome Extension.