Front-end optimizations for web applications

Font Sola, Octavi
Curs 2012-2013

Director: DAVINIA HERNÁNDEZ-LEO
GRAU EN ENGINYERIA EN TELEMÀTICA

Treball de Fi de Grau
## Contents

Abstract

<table>
<thead>
<tr>
<th>Language</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>English</td>
<td>7</td>
</tr>
<tr>
<td>Català</td>
<td>7</td>
</tr>
<tr>
<td>Castellano</td>
<td>7</td>
</tr>
</tbody>
</table>

Introduction

1 Web Traffic Shape

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>The need for speed</td>
<td>11</td>
</tr>
<tr>
<td>A typical Website</td>
<td>12</td>
</tr>
<tr>
<td>Summary</td>
<td>14</td>
</tr>
</tbody>
</table>

2 Network layer

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>More bandwidth, faster surfing?</td>
<td>15</td>
</tr>
<tr>
<td>Test: Vary the bandwidth</td>
<td>16</td>
</tr>
<tr>
<td>Test: Vary the RTT</td>
<td>16</td>
</tr>
<tr>
<td>Latency in perspective</td>
<td>17</td>
</tr>
<tr>
<td>Summary</td>
<td>23</td>
</tr>
</tbody>
</table>

3 Transport Layer

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Three-Way handshake (3WHS)</td>
<td>25</td>
</tr>
<tr>
<td>TCP Fast Open</td>
<td>27</td>
</tr>
<tr>
<td>Congestion Avoidance and Control</td>
<td>30</td>
</tr>
<tr>
<td>Flow Control</td>
<td>30</td>
</tr>
</tbody>
</table>
CONTENTS

Slow Start ........................................... 31
Slow Start Restart ................................. 32
Congestion Avoidance ............................. 32
Fast Recovery ........................................ 32
Head of line blocking .............................. 33
Summary ............................................... 33
Chapter references ................................. 33

4 Application Layer ................................. 35
Evolution of HTTP ................................... 35
HTTP 0.9 (1989-1991) ............................. 35
HTTP 1.0 (1991-1999) ............................. 36
HTTP 1.1 (1999-present) ........................... 37
HTTP 2.0 (2014-onwards) ......................... 37
General optimizations ............................. 38
HTTP 1.1 in detail .................................... 38
  Keep-alive connections ......................... 38
  HTTP pipelining .................................. 40
  Multiple TCP connections ....................... 41
  Domain sharding ................................ 42
  Caching and validation mechanisms .......... 44
  Resource compression .......................... 47
  Stylesheet and Script minification .......... 48
  Concatenation and spriting .................... 49
  Resource inlining ............................... 50
  HTTP 1.1 performance rules ................... 50
  Headers and cookies ............................. 51
HTTP 2.0 in detail ................................ 51
  Overview ......................................... 51
  Layer abstraction ................................ 52
  Binary framing .................................. 53
# CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiplexing</td>
<td>54</td>
</tr>
<tr>
<td>Prioritization</td>
<td>55</td>
</tr>
<tr>
<td>Header compression</td>
<td>55</td>
</tr>
<tr>
<td>Server push</td>
<td>55</td>
</tr>
<tr>
<td>Deployment</td>
<td>56</td>
</tr>
<tr>
<td>HTTP 2.0 performance rules</td>
<td>57</td>
</tr>
<tr>
<td>Summary</td>
<td>58</td>
</tr>
<tr>
<td>Chapter references</td>
<td>58</td>
</tr>
</tbody>
</table>

## 5 Presentation Layer

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3rd party snippets</td>
<td>60</td>
</tr>
<tr>
<td>Summary</td>
<td>61</td>
</tr>
<tr>
<td>Chapter references</td>
<td>61</td>
</tr>
</tbody>
</table>

## 6 Performance Assessment on LdShake

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Setting up the testing environment</td>
<td>63</td>
</tr>
<tr>
<td>LdShake architectural overview</td>
<td>66</td>
</tr>
<tr>
<td>Initial assessment</td>
<td>66</td>
</tr>
<tr>
<td>Login view</td>
<td>66</td>
</tr>
<tr>
<td>First steps view</td>
<td>68</td>
</tr>
<tr>
<td>Browse view</td>
<td>68</td>
</tr>
<tr>
<td>New LdShake view</td>
<td>68</td>
</tr>
<tr>
<td>Performance optimizations</td>
<td>68</td>
</tr>
<tr>
<td>Measuring</td>
<td>70</td>
</tr>
<tr>
<td>Summary</td>
<td>72</td>
</tr>
</tbody>
</table>

## Conclusions

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ending thoughts and future research</td>
<td>79</td>
</tr>
</tbody>
</table>

## Bibliography

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bibliography</td>
<td>81</td>
</tr>
</tbody>
</table>
Abstract

English

This Bachelor's Final Project takes a look at the current state of web applications and analyzes how physical links and the protocol stack impact their performance. This analysis presents various strategies to optimize the transport, application and presentation layers. It also introduces HTTP 2.0 and discusses how it will make the web faster. Finally, a performance assessment is made on an actual web application, LdShake (Hernández-Leo et al. [2011]). This assessment will offer practical insights on how front-end performance analysis are done and which kinds of optimizations yield the best results.

Català

Aquest treball de fi de grau realitza un estudi sobre l’estat actual de les aplicacions web i analitza quin impacte tenen els protocols de comunicació sobre el rendiment d’aquestes. El treball presenta diverses estratègies per optimitzar les capes de transport, aplicació i presentació. També introduceix HTTP 2.0 i explica com la nova versió del protocol ajudarà a accelerar la web. Per acabar, es duu a terme un cas d’estudi amb l’aplicació web LdShake (Hernández-Leo et al. [2011]). Aquest estudi serveix de guia per analitzar el rendiment d’una aplicació des del punt de vista del front end, i també per determinar quines optimitzacions donen millors resultats.

Castellano

Este trabajo de fin de grado realiza un estudio sobre el estado actual de las aplicaciones web y analiza qué impacto tienen los protocolos de comunicación sobre el rendimiento de estas. El trabajo presenta diversas estrategias para optimizar las capas de transporte, aplicación y presentación. También introduce HTTP 2.0 y explica como la nueva versión del protocolo ayudará a acelerar la
web. Para terminar, se lleva a cabo un caso de estudio con la aplicación web LdShake (Hernández-Leo et al. [2011]). Este estudio sirve de guía para analizar el rendimiento de una aplicación desde el punto de vista del front end, y también para determinar cuáles son las optimizaciones que dan mejor resultado.
Introduction

This project was born due my interest in web operations. Web operations is far from a well-defined field. It encompasses understanding of networks, routing, switching, firewalls, load balancing, high availability, disaster recovery, TCP, UDP, NOC management, hardware specifications, UNIX understanding, web servers, caches, databases, storage infrastructure, cryptography, algorithms, trending and capacity planning (Allspaw and Robbins [2010] p22). Since the field is so broad, and the number of problems to tackle so high, actually defining the scope of this project has not been easy.

The original purpose of the project was to pursue an investigation regarding scalability issues and patterns to provide solutions in that regard, but as I got deeper in the subject, the better I understood that scalability is a fickle issue, deeply tied to each platform’s architecture.

Reading about the subject, I came across High Performance Websites, a book centered on front-end performance. I was surprised about his claims that focusing on the front-end was the best way to improve the speed of a website (as much as 80%) (Souders [2007], p. xi).

Even though the subjects fundamentally tackle different problems: serving more users versus serving them faster, I found the second approach much more feasible (from a hardware point of view) and relevant to a wider range of products, because not everyone has a million users, but anyone can appreciate a faster service.

The project will try to answer the following questions:

- What causes slowness in web applications?
- What can we do about it?
- Are the changes worth it?

The project has been divided in 6 chapters:

On the first chapter, we will sketch the typical current web application and dissect it. This will help us get an idea of what has to be delivered to the user and why, or why not, the underlying pipes are optimized for this kind of traffic.
On the second one, we will analyze what kind of networking links our users have. The latency bottleneck will be introduced in this chapter.

On the third, we start dealing with the transport layer. How does TCP work, and what latency does it introduce to the connection will be answered here.

On the forth, we talk HTTP. I will do a quick tour on its evolution, on how it was supposed to be used, how it is used nowadays, and why those new use cases are really pushing the envelope. They are pushing it so hard, in fact, that a new version of HTTP (2.0) is on the works. We will also talk about that.

On the fifth we talk about browser rendering. How does the browser schedule the resources, how does it construct the render tree of a webpage and what can we do to help it work faster.

Finally, on our 6th chapter, we will perform and performance assessment on a real web application: LdShake (Hernández-Leo et al. [2011]). This chapter will serve as a wrap-up to the previous ones and a demonstration that, indeed, front-end performance does improve our users’ experience.
Chapter 1

Web Traffic Shape

In this initial chapter we will discuss the typical structure of a current webpage and the kind of resources it employs. It is very important that we are aware of what kind of traffic we must deliver to our clients to optimize our tools for it. We will also talk about the need for speed as a necessity to create compelling user experiences.

The need for speed

In a world where fast-paced developments are the norm, it may seem that implementing features quickly is more important than implementing them efficiently. Still, as true as this may be, user interaction does make or break a lot of apps. The usability of an app may very well determine its success and the willingness of a user to repeatedly use that service.

Speed and fast response times are one of the key aspects of usability engineering, as stated by Jacob Nielsen, one of the most well-regarded experts in that matter. From his Usability Engineering book (Nielsen [1993], ch. 5):

The basic advice regarding response times has been about the same for thirty years (Miller [1968]; Card et al. [1991]):

- 0.1 second is about the limit for having the user feel that the system is reacting instantaneously, meaning that no special feedback is necessary except to display the result.
- 1.0 second is about the limit for the user’s flow of thought to stay uninterrupted, even though the user will notice the delay. Normally, no special feedback is necessary during delays of more than 0.1 but less than 1.0 second, but the user does lose the feeling of operating directly on the data.
\begin{itemize}
  \item \textbf{10 seconds} is about the limit for \textit{keeping the user’s attention} focused on the dialogue. For longer delays, users will want to perform other tasks while waiting for the computer to finish, so they should be given feedback indicating when the computer expects to be done. Feedback during the delay is especially important if the response time is likely to be highly variable, since users will then not know what to expect.
\end{itemize}

From the citation we can infer that any interaction should finish in less than 0.1 to keep the app snappy and page loads should take about a second.

Faster pages generate better user engagement which, at the end of the day, translates to higher revenues. Therefore, if we think of speed as a feature, maybe skimping on it isn’t worth it after all.

\section*{A typical Website}

http://httparchive.org is a great resource of updated data about the current state of the web. It scans the top 10,000 ALEXA sites and analyzes how many resources they employ, their size, how long do they take to load, and other interesting data. Apart from the data they showcase in their web, it is also possible to download it and make your own statistics and analysis.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{average_webpage_size.png}
\caption{average webpage size, from http://httparchive.org}
\end{figure}

In figure 1.1 we can see the current average webpage (May 15 2013). It’s roughly 1.5MB in size. Images account for a 60% of the size, and scripts for another 20%. 
JPEG is the most popular image format, followed by PNG and GIF. Flash objects are the biggest in size (on average).

From this first 3 figures it is pretty clear that the current average webpage (as of May 15) is about 1.5MB in size, and is composed of 92 requests. The majority of its content is images (60+%) and Javascript.

Each individual resource also tends to have a rather small size. HTML, Javascript and CSS don’t usually go over 15kB, and it is only images and flash content that makes heavy use of bandwidth.

We can see from the detailed view of transfer size per object that the amount of Javascript, HTML and CSS is roughly the same as it was a year before (stats
from 15/04/12 to 15/05/13) but, while the number of image requests is also constant, the transfer size has steadily increased. This could be due to added support for new high density screens (tablets and laptops).

Summary

In this chapter, we have taken a glimpse at the importance of interactivity in websites. From the data retrieved from httparchive we can also see that javascript is indeed very important for the actual web.

This reliance in javascript and technologies like AJAX allows us to create single-page websites more easily, allowing us to change its content dynamically.

The rise in interactivity, multimedia content and real time services puts a great strain in the current network stack, which was originally designed to transfer static content.

In the next chapters we will talk about how the network, the transport and the application layer delay the delivery of a webpage.
Chapter 2

Network layer

In this chapter I will discuss how bandwidth and latency affect the performance of a webpage.

More bandwidth, faster surfing?

Even though it may seem that the key factor to speed up web surfing should be bandwidth, it has been proven that it does not matter that much (Belshe [2010]). A typical webpage instantiates multiple HTTP connections to transmit small resources, so the TCP connections tend to be short and bursty, which is not what TCP is optimized for. This means that the bottleneck when delivering webpages is basically a matter of latency.

Improving latency, in contrast to improving bandwidth capacity, is a very hard thing to do. Latency is limited by the propagation speed of the signal on the wire, and those speeds are already pretty close to the speed of light, so the only way it could be reduced is by deploying shorter links.

Latency may be theoretically limited by the distance between a pair of endpoints, but in practice, the connection’s last mile is also frequent bottleneck. The technology employed by the user’s connection will greatly influence the round trip time. The variance can be as high as 2 orders of magnitude from the best connections to the worst (fibre vs GPRS).

In the following paragraphs, I will show a series of tests performed on Belshe [2010] where we can clearly identify this bandwidth bottleneck. I will also end the chapter talking about which kind of connections we can expect from our users (relevant to perform realistic performance tests on our websites) and how mobile connections exacerbate this latency issue.
Test: Vary the bandwidth

In this test a fixed RTT value is used (60ms), while the bandwidth has been increased. The test has been performed with 25 pages of the Alexa top.

<table>
<thead>
<tr>
<th>Bandwidth</th>
<th>Page Load Time via HTTP</th>
</tr>
</thead>
<tbody>
<tr>
<td>1Mbps</td>
<td>3106ms</td>
</tr>
<tr>
<td>2Mbps</td>
<td>1950ms</td>
</tr>
<tr>
<td>3Mbps</td>
<td>1632ms</td>
</tr>
<tr>
<td>4Mbps</td>
<td>1496ms</td>
</tr>
<tr>
<td>5Mbps</td>
<td>1443ms</td>
</tr>
<tr>
<td>6Mbps</td>
<td>1406ms</td>
</tr>
<tr>
<td>7Mbps</td>
<td>1388ms</td>
</tr>
<tr>
<td>8Mbps</td>
<td>1379ms</td>
</tr>
<tr>
<td>9Mbps</td>
<td>1368ms</td>
</tr>
<tr>
<td>10Mbps</td>
<td>1360ms</td>
</tr>
</tbody>
</table>

Table 2.1: PLT vs Bandwidth

As we can see, once we have hit the 5–6Mbps mark, we get diminishing returns for every added megabit. It is appalling that a 10Mbps connection can only use 1.6Mbps of effective bandwidth to download a page.

Test: Vary the RTT

In this test the bandwidth is fixed at 5Mbps and the RTT varied from 240ms to 0ms.

<table>
<thead>
<tr>
<th>RTT</th>
<th>Page Load Time via HTTP</th>
</tr>
</thead>
<tbody>
<tr>
<td>240ms</td>
<td>3964ms</td>
</tr>
<tr>
<td>220ms</td>
<td>3707ms</td>
</tr>
<tr>
<td>200ms</td>
<td>3418ms</td>
</tr>
<tr>
<td>180ms</td>
<td>3151ms</td>
</tr>
<tr>
<td>160ms</td>
<td>2874ms</td>
</tr>
</tbody>
</table>
Table 2.2: PLT vs RTT

We can see that reducing the RTT gives us a linear increase in performance, in contrast to increasing bandwidth. It is still surprising to see that, even with an RTT of 0, we can only use about half the available bandwidth (2.5Mbps) to transmit the web. This is certainly an improvement over the 1.5Mbps available with an RTT of 60ms, but very far from the full capacity of the link. This performance loss is due to the way the transport and application layer work, which we will discuss in the next chapters. The browser renderer scheduler also affects the perceived performance of the page, which is another aspect we will discuss.

Latency in perspective

To put the latency issue in perspective, I will extract some data from the periodical broadband reports issued by the FCC in the US. Specifically, I will refer to the February 2013 (FCC [2013]), the July 2012 (FCC [2012]) and the August 2011 (FCC [2011]) reports.

Table 2.3: Latency broadband USA
CHAPTER 2. NETWORK LAYER

Figure 2.1: Page load time vs bandwidth, from Belshe [2010]

Figure 2.2: Bandwidth percentage improvement, from Belshe [2010]
Figure 2.3: Effective bandwidth usage, from Belshe [2010]

Figure 2.4: Page load time vs RTT, from Belshe [2010]
Figure 2.5: RTT percentage improvement, from Belshe [2010]

Figure 2.6: Effective bandwidth usage, from Belshe [2010]
As we can see, those numbers remain basically unchanged in 2.5 years, while the average subscribed speed has gone from 11.1Mbps (August 2011) to 15.6Mbps (February 2013). Those are US numbers, but they are a good reflection of the overall tendency.

This chart is also interesting as we can see how bandwidth influences latency very little. Only DSL shows any kind of significant variance, and it can still be counted among the 10s of milliseconds.

In the mobile space, things are much worse. If we take data from the Grigorik [2012a]:

> Users of the Sprint 4G network can expect to experience average speeds of 3Mbps to 6Mbps download and up to 1.5Mbps upload with an average latency of 150ms. On the Sprint 3G network, users can expect to experience average speeds of 600Kbps - 1.4Mbps download and 350Kbps - 500Kbps upload with an average latency of 400ms.

So mobile has very, very bad latency. And mobile is key. (“We talk about ‘mobile first’ in 2012, but we want to be ‘mobile best’ in 2013” (Kanalley [2013]) as stated by Facebook Vice President of Partnerships Dan Rose).

Putting some worldwide numbers on latency:

Avg RTT to Google in 2012 (Grigorik [2012b]):

- Worldwide: 100ms
• US: ~50-60ms
### Summary

As we have seen in this chapter, decreasing latency is the best way to decrease page load times. Unfortunately, we have already reached the physical limits of our networking technologies, so we can’t hope to improve latency by improving the physical layer. This means that all our efforts should be directed in creating protocols tailored for low latency.
Chapter 3

Transport Layer

In this section I will discuss the mechanisms that TCP uses and their effects on webpage load times. I will also propose some deployable fixes to increase the performance of this layer.

TCP is the transport protocol over which HTTP works. TCP provides a virtual, reliable, end-to-end pipe between hosts processes. Since TCP works over an unreliable stack, it provides mechanisms for:

- retransmission of lost data
- in-order delivery
- congestion control and avoidance
- data integrity
- ...

TCP is very convenient for accurate delivery of data, but the failsafe mechanisms it uses, introduce delay. In this section, we will discuss them further, with special emphasis on their interaction with the typical HTTP flow.

Three-Way handshake (3WHS)

Since a TCP stream is not stateless, it is necessary to perform a handshake between the client and the server before any data can be transmitted. It is also important to note that, since the application layer and the transport layer are independent, there is no way around this. The connection schema is as follows:

- client sends SYN, with random sequence number and optional TCP flags.
- server sends SYN ACK, with its own flags and options.
• client sends ACK. At this point application data can be sent already.

It is important to note that before any data can be sent a full roundtrip has been spent. This roundtrip delay does not depend on bandwidth but on latency, and latency is basically capped by the length of the link and the speed of light.

So, to summarize, opening a connection has a variable RTT cost that must be taken into account, in the form of latency. We want to reuse TCP connections as much as possible (that is why HTTP1.1 introduced keep-alive by default, instead of connection per resource).

If we take a look to the handshake process under HTTPS (that is, HTTP over a TLS authentication layer), the delays are even more dramatic, since 2 additional RTT are needed (1 if the resumed version can be used).

The TLS needs one RTT for the server to send it’s public certificate to the client. When the client received it, the pubkey is checked against it’s CA, and if it is deemed valid (authentication step), the client will decide on a key to encrypt the
communications (symmetric-key encryption). This key will be encrypted with the public key of the server and sent during the second RTT, along with the encryption schemas? (AES, DES...) Finally, the server will accept some configuration, send an ACK and, after that, the data can be sent.

![Three-way TLS resumed handshake](image)

**Figure 3.3**: Three-way TLS resumed handshake, from Grigorik [2013c], ch. 4

If we have already established one encrypted connection with the server, we already hold a copy of its public key, so the first round of the communication is not needed. This is why browsers hold multiple HTTPS connections to the same host until the first one has been established, so they can prevent a roundtrip from happening.

Even if establishing a TLS connection is costly, it has many advantages over an unencrypted connection. Apart from the security issue, since the data sent through HTTPS is encrypted and, thus, obscured, it is possible to use other protocols than HTTP over TLS without being blocked by firewalls or proxies. We will discuss why this is convenient during our talk about HTTP 2.0 and SPDY.

**TCP Fast Open**

This proposal enables the carrying of data within a SYN / SYN ACK packet, thus saving up to one full round trip time compared to standard TCP 3WHS. It was proposed by Google employees in 2011 (Radhakrishnan et al. [2011]), and its RFC is currently being developed at Chu et al.. It has been implemented in Linux kernel 3.7+, for both client and servers.

Similar proposals have been made (see Radhakrishnan et al. [2011]) but they haven’t gained much traction, required too many changes on the implementation (and socket API), introduced security issues or were more costly (computationally). Another similar alternative currently implemented by browser vendors is the PRECONNECT state, where the browser preemptively opens a TCP connection to the most visited sites by the user, avoiding the DNS resolution and
TCP 3WHS. Since the probability that the user will visit one of those sites is high, the perceived performance of the browser will be better, and the drawbacks small (Grigorik [2013d]).

The current TCP implementation already allows sending SYN packets with data, but the server is forbid to reply to them until the connection has been established. This is so mainly for security purposes. If the server could reply to a request without a previous handshake it would be open to attacks such as:

**SYN Flood / Server Resource Exhaustion** By sending SYN packets with requests, the server should also handle the load to compute the response, which may be quite high.

**Amplified Reflection Attack** By spoofing the IP of the requests, the server would send the responses against any victim. This way, any server implementing TFO could be used as a part of a botnet by anyone.

To prevent this 2 issues, while allowing a fast TCP connection and keeping the changes to the TCP protocol to a minimum the following 2-phase protocol has been defined:

**Requesting a TFO security cookie**

1. The client sends a SYN packet with a Fast Open Cookie Request in the TCP option field.
2. The server generates a cookie bytestring by encrypting the client address with some symmetric-key cipher (currently an 8byte cookie, encrypted with AES-128). The server responds with a SYN-ACK that includes the generated cookie in the TCP option field.
3. The client can cache this cookie to Fast-Open connections to the server. It is also important to note that this procedure has established a valid TCP connection that may be used to send data (getting the cookie is not a performance drawback from standard TCP).

**Using a TFO cookie**

1. The client sends a SYN with a cached Fast Open cookie (as TCP option) along with the application data.
2. The server validates the cookie by decrypting it and matching it with the client IP address (or viceversa).
   a. If the cookie is valid: The server sends SYN-ACK that acknowledges the SYN and the data. The data is delivered to the server.
   b. If the cookie is not valid: The server sends a SYN-ACK that only acknowledges the SYN, not the data. The data is dropped. Connection will fallback to a regular 3WHS.
3. If the SYN data was accepted, the server may transmit additional response data segments to the client before receiving its first ACK.
4. The client sends an ACK acknowledging the server SYN. If the client’s data was dropped, it is sent with the ACK (as a normal 3WHS connection would).
5. The connection has been established and proceeds as a normal TCP connection.

Figure 3.4: TFO connection overview, from Radhakrishnan et al. [2011]

The encrypted cookie is the main mechanism against the 2 previously detailed vulnerabilities.

**SYN Flood / Server Resource Exhaustion** The server will not process any request with a foul or nonexistent cookie, but it must be noted that anyone could request a valid cookie. The proposed strategy to prevent resource exhaustion is to have a limited number of pending TFO requests, that is, connection requests that haven’t been fully established. If the total number of TFO requests should exceed that number, the server would deactivate TFO and switch to 3WHS. The protocol is still vulnerable to SYN flooding, but not more than standard TCP is.
Amplified Reflection Attack  Since the cookie consists of the encrypted IP with a secret key (that has to be periodically revoked), it is not easy to spoof requests to overload a targeted victim. This attack can only take place if the key is compromised or the attacker steals a valid cookie from the target. The proposal also argues that if a system has been compromised to such an extent, there will be far more damaging ways to attack it.

Congestion Avoidance and Control

In the early 80s, when the internet was still called ARPANET, the congestion collapse problem was observed for the first time. As stated by John Nagle (RFC 896, see Nagle):

> Congestion control is a recognized problem in complex networks. We have discovered that the Department of Defense’s Internet Protocol (IP), a pure datagram protocol, and Transmission Control Protocol (TCP), a transport layer protocol, when used together, are subject to unusual congestion problems caused by interactions between the transport and datagram layers. In particular, IP gateways are vulnerable to a phenomenon we call "congestion collapse", especially when such gateways connect networks of widely different bandwidth.

In those early times this was not much of a problem, since most nodes had uniform bandwidth and excess capacity, but that didn’t hold true for long. This is the reason TCP has multiple built-in mechanisms to regulate the rate in which data can be sent:

- flow control
- congestion control
- congestion avoidance

We will explain those mechanisms further, paying special attention on the aspects that hurt HTTP performance the most.

Flow Control

Flow control is a mechanism that prevents the sender from overwhelming the receiver by sending more data than it can handle. To control the data rate, each side advertises a receive window ($rwnd$) in its ACK. This way, it can be dynamically adjusted based on the processing speed of the sender and the receiver. In a web stream, it is typically the client window the bottleneck factor.
Slow Start

Flow control prevents the sender from overwhelming the processing power of the receiver, but it does nothing to prevent the overload of the underlying network. The available bandwidth of the link is unknown to both parties, so we need a mechanism to adapt the throughput of the connection to the available resources as fast as possible. This is where slow start comes into play.

Each sender has an internal congestion window size \((cwnd)\) variable, that starts small and grows as long as packets are acknowledged. The initial value of the \(cwnd\) window is set by the operating system and has seen much debate over the years. In the original RFC for TCP (Postel [1981]) it was set to one. Later one, in RFC 2581 (April 1999) (Stevens et al. [1999]), its value was increased to 4 and, recently, in RFC 6928 (April 2013) (Chu et al. [2013]) has been set to 10. The initial window size limits the effective bandwidth of links in short-lived connections (such as HTTP ones tend to be), so bigger limits yields better performance. In fact, the suggestion that the \(cwnd\) should be increased to 10 was noted in Herbert et al. [2009]. The paper compares latency using different \(cwnd\), and discovers that a \(cwnd\) of 10 yields a +13% performance over a \(cwnd\) of 1 (against a +8% of a \(cwnd\) of 4).

Each time an ACK is received, the number of packets sent by the sender is doubled, so the \(cwnd\) window grows exponentially.

The slow start phase ends when \(cwnd\) goes over a OS fixed threshold called ssthresh or a packet is lost. In the first case, TCP enters congestion avoidance mode, while in the latter case it switches to fast recovery mode.

Slow-start is an important phase in web performance because it limits the available bandwidth of the link at the start of the connection, and since a lot of HTTP connections are short and bursty, they may never get to use the full link throughput.

![Figure 3.5: TCP Slow Start, from Grigorik [2013c], ch. 2](image)
CHAPTER 3. TRANSPORT LAYER

Slow Start Restart

This mechanism resets the cwnd of an idle TCP connection. The reasoning behind this behaviour is that the network conditions may have changed while the connection has been idle, so it is safer to start transmitting with a lower window. Since SSR has a negative impact on HTTP keep-alive connections, it is often disabled on servers:

```
$> sysctl net.ipv4.tcp_slow_start_after_idle
$> sysctl -w net.ipv4.tcp_slow_start_after_idle=0
```

Increasing the initial cwnd size it’s an easy way to improve the performance of the server. Any Linux kernel above 2.6.39 already works with the new default value of 10. Kernel 3.2+ also has other updates like proportional rate reduction for TCP (congestion mechanism).

Congestion Avoidance

If we continuously doubled the cwnd, we would soon be sending packets over the capacity of the network and start losing them. To avoid this undesirable behaviour, while slowly approaching the bandwidth limit of the network a new threshold, called Slow Start Threshold (ssthresh), is introduced. ssthresh is set to half the cwnd value when the connection last suffered a packet loss.

Once ssthresh is reached, the cwnd will increase linearly. Instead of doubling the window each RTT, cwnd will only increase by one. If a packet is lost or we receive a triple duplicated ACK, TCP enters Fast Recovery mode.

Fast Recovery

This is the phase where most TCP implementation differ. The first implementation of TCP (Tahoe) didn’t have any fast recovery mechanism and each loss triggered slow start. Shortly after, TCP Reno was introduced with a fast recovery mechanism, and every other TCP implementation since has followed that schema. The current implementations of TCP on Linux (CUBIC) (Rhee and Xu) and Windows (Compound) (Tan et al. [2006]) work in a similar fashion, but are much more aggressive, which enables them to get optimal performance from the link much faster. For a more in-depth comparison between current TCP implementations I recomend the paper Performance Analysis of Modern TCP Variants [Symposium 2010]. It does a very interesting comparison between CUBIC, Compound and New Reno and shows how fast and fair this implementations are.

Fast Recovery resends the lost packet and sets the cwnd to cwnd / 2. ssthresh is also set to cwnd / 2. From this point we enter congestion avoidance phase once again.
Head of line blocking

Since TCP ensures that the packets will be delivered in order, if a packet is lost, all subsequent packets must be held in a buffer until the lost packet has been recovered. This introduces jitter, which may be fatal in certain kinds of applications, like video or audio streaming or online gaming. For this reason, TCP is not recommended for realtime applications.

Summary

As we have seen, for typical web traffic, the delays in setting up a TCP connection are the main factor that hurt our performance from a transport layer perspective. To mitigate this we may:

- **Increase the initial congestion window** to 10, by upgrading the kernel.
- **Disabling Slow Start Restart**, so already established connections don’t have to go through that initial phase.
- **Enable window scaling**, so the connection is not limited by \( rwnd \) but by \( cwnd \).
- **Enable TCP Fast Open**, and data will be sent in the first rountrip.
- **Put servers closer**, so RTT are reduced. This usually means using a CDN to store resources.
- **Reuse established TCP connections**, by using keep-alive connections in HTTP.

Although TCP hasn’t changed that much since its origin, updating our servers to the latest kernels is the best way to ensure our services are delivering the best transport layer performance possible.

Chapter references

This chapter is based on Grigorik [2013c], ch. 2. Congestion avoidance, fast recovery and head of line blocking have been expanded with Kurose and Ross [2010], ch. 3.
Chapter 4

Application Layer

In this chapter we will talk about the origins of HTTP (0.9 / 1.0), its current state (1.1), and the future of the protocol (2.0). We will make special emphasis on the current strategies to deliver high performance through HTTP 1.1 and how HTTP 2.0 will (and is) changing that.

Evolution of HTTP

Let us start doing a quick summary to the milestones of the HTTP protocol.

HTTP 0.9 (1989-1991)

HTTP was first proposed by Tim Berners-Lee, the inventor of the World Wide Web, in 1991. In the first version of the protocol the requests were as follows:

- GET command, followed by a resource identification. Additional keywords could be added after a question mark.
- Request is terminated by a carriage return.
- Request is ASCII string.

The format of the server responses:

- Response in HTTP or simple plain text.
- Response is an ASCII stream.
- ISINDEX tag indicates id of other valid resources.
- Connection is terminated after response is sent.
Other protocol characteristics:

- Stateless
- Idempotent
- Able to refer to other resources

As we can see, the core of what constitutes HTTP has not changed much: it still is a protocol mainly targeted at transferring files and referring clients to other servers and resources.

More information about the implementation of HTTP 0.9 can be found in (World Wide Web Consortium (W3C) [2011]) and (World Wide Web Consortium (W3C) [1991]).

**HTTP 1.0 (1991-1999)**

As the WWW began to grow in numbers, the use cases for HTTP expanded and it soon became clear that the protocol would have to expand beyond the delivery of hypertext. The evolution from 0.9 to 1.0 was a very organic affair, developing from 1991 to 1995 in a rather anarchic way. It was in May 1996 that the RFC 1945 (Berners-Lee et al. [1996]) was published, describing the common usage of HTTP until now. The same RFC already states that it is no more than a memo:

“This memo provides information for the Internet community. This memo does not specify an Internet standard of any kind. Distribution of this memo is unlimited.”

From the RFC we can summarize HTTP 1.0 as follows:

- Request may contain multiple lines, separating the headers.
- Responses have a header with status information.
- Responses not limited to HTML.
- Encoding of the response could be declared in the headers.
- Responses could be compressed.
- Authorization mechanisms were added.
- Caching mechanisms were added.
- Connection is terminated after response is sent.

With those changes, HTTP 1.0 became a whole lot more useful than its predecessor 0.9, but we can still see 2 design flaws that will hurt efficient delivery of the content:
• Connections are not reused. This means, for every resource, having to to through the TCP 3WHS and Slow Start phases.

• Headers are not compressed. This one is big especially nowadays, when a lot of the data is sent through many small JSON requests. In those cases were the responses are very small, the overhead may be on the 80–90% range.

HTTP 1.1 (1999-present)

Six months after the publication of the HTTP 1.0 RFC, the first draft of what would become HTTP 1.1, RFC 2068, was published (Fielding et al. [1997]). The final version of the protocol was delayed until 2 years later, in June 1999, as RFC 2616 (Fielding et al. [1999]).

HTTP 1.1 was not a huge departure in functionality from 1.0, but it added a lot of features to improve its performance, the main of which were:

• keep-alive connections by default.
• chunked encoding transfers.
• byte range requests.
• additional caching mechanisms.
• transfer encodings.
• request pipelining.
• client cookies.

HTTP 2.0 (2014-onwards)

The seed of what will become HTTP 2.0 was planted in 2009, with the introduction of the SPDY protocol, developed at Google. SPDY is a protocol designed with minimal latency in mind, whose main technical goals are:

• multiplexing of many concurrent HTTP requests on a single TCP connection.
• reduced overhead by compressing headers.
• simple and efficient parsing of the message protocol.
• TLS as underlying transport protocol, for better security and compatibility.
• enable server initiated communications (push) with the client whenever possible.

SPDY (and HTTP2.0) do not plan to replace the HTTP syntax, but to provide better ways to take advantage of the underlaying network. The results so far are
promising: 39% - 55% speedup over HTTP, as stated by the introductory SPDY whitepaper (Belshe and Peon [2009]).

Even though the future is already here, with SPDY in its 3rd iteration, and deployments by Google (@[?), Twitter (Raghav [2012]), Facebook (Finley [2013]) and other major players like Amazon (Paul [2011]), it’s adoption rate is very low (0.9% of all websites) (Q-Success [2013]). Still, browser support is sizeable (54.19%) (CanIUse) and increasing (Rivera [2013]), which will hopefully convince more websites to make the switch.

General optimizations

HTTP 1.1 in detail

It soon became clear that HTTP 1.0 was not flexible enough to deliver the increasingly diverse media of the WWW. The protocol had too much overhead (one TCP connection per resource), didn’t support sending variable length transfers, or specific parts of a document, its cache mechanisms were primitive and there was no way to keep state of the clients. With all of this in mind, the 1.1 HTTP standard introduced:

- **keep-alive connections** to reuse established TCP connections for various HTTP requests.
- **chunked encoding transfers** to allow sending documents one piece at a time. This allows server to send documents with unknown length, which enables earlier flushing of dynamically generated content.
- **byte range requests** to allow partial transmissions of a resource. This is what allows pausing and restarting of downloads.
- **better caching and revalidation mechanisms** like the Cache-Control, If-none-match and ETag headers.
- **transfer encodings**. The tags Accept-Encoding and Transfer-Encoding allow both client and server to send the data in plaintext, compressed (gzip) or chunked.
- **request pipelining**, which enables sending multiple requests simultaneously.
- **client cookies**, which are included in the request headers and allow the server to keep track of the user session.

Keep-alive connections

One of the most significant changes in HTTP 1.1 was the introduction of persistent connections. Until then, each resource had to initiate a connection,
HTTP 1.1 in Detail

with its own handshake (+1 RTT) and go through the Slow-Start phase each time. This made sense originally, since webpages were hypertext only or the number of alternative resources was very low. Since this is not the case anymore, reusing TCP connections is a good way to make better use of the bandwidth.

Figure 4.1: HTTP with no keep-alive, from Grigorik [2013c], ch. 12

In this diagram we observe that the first case (sequential, no keep-alive) needs 284ms to transfer an HTML and CSS file. With keep-alive the number drops to 228ms. One less RTT. The savings can get much larger depending on the latency of the network and the congestion window of the server. As explained in the 4th chapter, small congestion windows on the server will make even small transfers like HTML files slower. In this case it is even more important to skip the Slow Start phase.

Keep-alive is so important that the feature was backported to HTTP 1.0 and it stands as one of the main reasons not to serve HTTP 1.0 requests.
CHAPTER 4. APPLICATION LAYER

Figure 4.2: HTTP with keep-alive, from Grigorik [2013c], ch. 12

HTTP pipelining

In a typical HTTP flow, a request can only be sent after receiving a response. This is a suboptimal way to transfer files, since the server sits idle for a full RTT between requests. To solve this problem, HTTP pipelining was introduced. With HTTP pipelining it is possible to send consecutive requests before receiving a response.

Figure 4.3: HTTP sequential pipelining, from Grigorik [2013c], ch. 12

As we can see from the image, the server has a FIFO queue that processes each
request sequentially and sends it to the client. We may even process the requests in parallel, like this:

![Diagram of HTTP parallel pipelining](image)

**Figure 4.4:** HTTP parallel pipelining, from Grigorik [2013c], ch. 12

If we pay close attention to the diagram, it is clear that even if the CSS is processed before the HTML, it is held until the HTML has been sent. This problem receives the name of **head of line blocking**, and it may severely degrade the performance perceived by the client if any of the scheduled requests hangs or suffers unexpected delays of any kind, since all other resources must wait for it to finish. It may also cause performance problems for the server, that has to buffer the already-to-be-dispatched responses until the first one is out (may lead to resource exhaustion). Better scheduling of the requests is also impossible, since the client does not know the cost of each.

HTTP pipelining also causes problems with incompatible intermediaries, that may drop the connection or serialize it.

Since the implementation of HTTP pipelining has been riddled with issues it is not even enabled by default in browsers. 2 approaches to this issue exist: implement multiplexing (HTTP 2.0 way), which allows interleaving of multiple requests in the same connection, or allow multiple TCP connections, which brings us to the next section.

**Multiple TCP connections**

Pipelining was a great idea (performance wise) poorly implemented so, to get a similar speedup, the HTTP standard allowed multiple parallel TCP connections per host.

Opening multiple TCP connections per page adds more complexity to the client, that has to deal with multiple sockets. It also consumes more CPU and memory resources, jeopardizes TCP fairness mechanisms and hurts battery life.

The HTTP 1.1 RFC says to limit persistent connections to 2 per server.
Clients that use persistent connections SHOULD limit the number of simultaneous connections that they maintain to a given server. A single-user client SHOULD NOT maintain more than 2 connections with any server or proxy. A proxy SHOULD use up to 2*N connections to another server or proxy, where N is the number of simultaneously active users. These guidelines are intended to improve HTTP response times and avoid congestion.

In practice, though, any browser will allow a maximum of 6 TCP connections. This magic number of 6 has been growing throughout the years and is basically a compromise between the overhead of establishing new connections and the degree of parallelism. Six connections tends to work well enough for most sites, but this is not the case for every site. In those cases, resources are served from multiple domains or subdomains, a technique that received the name of domain sharding.

Domain sharding

If 6 parallel connections are not enough, resources may be split into multiple domains. This forces a new DNS resolution on the client (more delay) and makes the site harder to manage to the admin, but may yield performance improvements, especially on heavy sites.

I have performed some tests on the flickr explore page (http://www.flickr.com/explore) to show how sharding is employed by a heavy website:

We can see from the chrome network inspector that images are downloaded from multiple domains, called farmX.staticflickr.com. Now we perform some dns lookups on those farms, to see where they resolve:

```
$ nslookup farm3.staticflickr.com
...    
Address: 77.238.160.184

$ nslookup farm4.staticflickr.com
...    
Address: 77.238.160.184

$ nslookup farm6.staticflickr.com
...    
Address: 77.238.160.184

$ nslookup farm8.staticflickr.com
```
Figure 4.5: Flickr domain sharding
Address: 77.238.160.184

$ nslookup farm9.staticflickr.com
...  
Address: 77.238.160.184

It is very clear now that the domains are only aliases to the same server. In fact, the domain is also behind a CDN. If we try to lookup the address from the United States, for example, it resolves to a different ip:

$ nslookup farm6.staticflickr.com
...  
Address: 64.90.63.203

Different cnames are used to download a larger stream of images from the same server in parallel. In this particular example, employing this technique makes a lot of sense, since flickr explore is an infinite-scrolling page with photographs. With so many images to serve, more connections will help to lessen the load times of the page and improve the user experience. The usage of a CDN to distribute the content also brings us to the next point.

CDN

If we do not want to deal with sharding ourselves, or we are interested in lowering the latency for our users, using a CDN to deliver our static resources is a popular and well proven choice.

A CDN will store and distribute the content we feed them all around the world, lowering the latency median. You won’t have to take care of sharding or caching, and traffic spikes will be more manageable, since we are only dealing with the generation of the dynamic content.

CDNs are very widely used nowadays, and have become increasingly necessary to deal with the popularity of multimedia streaming sites, like YouTube.

Caching and validation mechanisms

Having a fast fresh view of our applications is important, especially for new users, but existing users can get even better performance of a web application, if properly configured. HTTP cache mechanisms provides us tools to store resources in the client, which can decrease greatly the number of requests to our server, making the web load faster and offloading our servers.

In order of priority, those are the ways a server can specify the freshness of a resource:
• Cache-Control: no-store header.
• Cache-Control: no-cache header.
• Cache-Control: must-revalidate.
• Cache-Control: max-age.
• Expires header.
• No cache header. Let the cache determine by itself.

**Cache-Control: no-store, no-cache and must-revalidate**

**no-store** A response marked as *no-store* can’t be copied to the cache and must be forwarded directly to the client. The purpose of *no-store* is to prevent sensitive data to be leaked through cache tampering. Since some cache implementations may ignore the *no-cache* header, it may be necessary to enable *no-store* to ensure that the content is fresh.

**no-cache** A response marked as *no-cache* may be copied to the cache, but must be revalidated with the server each times it is requested.

**must-revalidate** Some caches are configured to serve stale (expired) content. The *must-revalidate* header forces those caches to recheck with the server whether the resource is fresh or not.

**Cache-Control: max-age and Expires**

**max-age** Seconds until the resource is considered stale. *max-age* uses relative time measures and is preferred over *Expires*.

**Expires** Absolute expiration date of the resource. Depends on the computer clock set correctly, so it is not as reliable as *max-age*.

**Heuristic expiration**

If no cache is specified, the client may try some heuristics to determine the expiry date of the resource. The *LM-Factor* algorithm is a pretty extended approach:

- calculate the time between the request and the *Last-modified* header of the resource.
- Multiply this delta by some factor, like 0.2.
- The result is the estimated *max-age* of the resource.

To prevent the heuristic to grow too large, usually upper bounds are placed (a day or a week are popular choices). Still, since this kind of approaches are not enforced nor standard, they should not be relied upon. This is why cache-headers should always be specified, to guard ourselves from erratic client behaviours.
Validation headers

An expired document does not necessarily mean it’s gone stale, it means we can’t assure it’s fresh. To avoid requesting the resource anew HTTP can use conditional methods. Conditional methods allow the server to return a 304 Not Modified response to the client, instead of the whole resource, which leads to a reduction in transfer times and server load.

The 2 conditional headers used in cache revalidation are:

**If-Modified-Since <date>** Perform GET if resource has been modified since the specified date.

![Request with If-Modified-Since headers](image)

**If-None-Match <tag>** Perform GET if resource *ETag* does not match the ones in the header. If not, return 304 Not Modified response. *Etags* are labels attached to a resource with a unique identifier to represent the resource contents.

Usually, If-Modified-Since revalidation is good enough, but there are some cases when it falls short:

- Rewritten resources that contain the same data. Even if the content is the same, the modification date on the filesystem has been changed.
- Minor changes that do not warrant propagation.
• Resources with sub-second update intervals (like realtime monitoring). IMS has a 1 second granularity, so it could not be used in those cases.

To overcome these problems, HTTP 1.1 introduced Entity Tags (ETags). Etags are arbitrary labels to identify versions of a resource and take precedence over Last-Modified headers. It is very important to configure Etags correctly if requests are handled by a cluster. Apache and IIS webservers use an ETag format that is system dependant, so the same cluster may very well have different Etags for the exact same resource. To avoid this undesirable behaviour, Etags should be either manually configured or removed entirely.

![Figure 4.7: Request with If-None-Match headers, from Gourley et al. [2002], ch. 7](image)

**Resource compression**

HTML does not allow header compression, but it certainly enables mechanisms to compress its payload. Resource compression is widely deployed for both clients and servers, fairly light on CPU resources and a highly effective way to reduce the transmitted byte count.

To manage encoding negotiation, HTTP clients inform the server with the header Accept-Encoding, with a list of available compression methods.

Accept-Encoding: gzip, deflate

A server returning a compressed response, will indicate the algorithm in its Content-Encoding header:
CHAPTER 4. APPLICATION LAYER

Content-Encoding: gzip

Not all resources benefit equally from being compressed though. Multimedia content has its own set of specialized compression algorithms and recompressing it with gzip (or any other general purpose algorithm) may even lead to bigger file sizes. This is why only text based content (HTML, CSS, Javascript and plaintext) should be compressed. Multimedia file format may be a key decision for some web applications to keep its bandwidth expenditure at bay, although the winners nowadays seem clear: mp3 or AAC for audio, jpeg and png for images and h.264 for video. Still, there are emerging alternatives like WebP (Google Inc. [2012]) for images or VP8 (Bankoski et al. [2011]) for video. In the video side, the successor of h.264, HEVC, has recently been published (June 7th) as a final standard (Sector and Itu [2013]), so it is reasonable to expect a migration to this new, more efficient (Ohm et al. [2012]) (+35%) standard in the following years.

Multimedia content aside, gzip compression (most widely supported format) usually yields 70% reductions in file size so there is no reason not to use it.

<table>
<thead>
<tr>
<th>File type</th>
<th>Uncompressed size</th>
<th>Gzip size</th>
<th>Savings</th>
</tr>
</thead>
<tbody>
<tr>
<td>Script</td>
<td>3.277 bytes</td>
<td>1076 bytes</td>
<td>67%</td>
</tr>
<tr>
<td>Script</td>
<td>39.713 bytes</td>
<td>14.488 bytes</td>
<td>64%</td>
</tr>
<tr>
<td>Stylesheet</td>
<td>968 bytes</td>
<td>426 bytes</td>
<td>56%</td>
</tr>
<tr>
<td>Stylesheet</td>
<td>14.122 bytes</td>
<td>3.748 bytes</td>
<td>73%</td>
</tr>
</tbody>
</table>

Stylesheet and Script minification

Even though it may seem like the road to axing bytes is over after the compression step, we can still do better. Both javascript and css files tend to include a lot of white space, comments and other unnecessary characters that may be stripped without changing its functionality.

There are minification tools that perform this process automatically, so the development version can be as clean and readable as possible while keeping the production version size at a minimum. The Google Closure compiler (https://developers.google.com/closure/compiler/) or UglifyJS (https://github.com/mishoo/UglifyJS2) are popular choices for javascript minification, while the YUI Compressor (http://yui.github.io/yuicompressor/) seems to be the tool of choice for CSS minification.

I have performed some minification and compression test on the jquery to toss some data over which kind of size reduction might be expected:
As we can see, minifying can be as powerful as compressing. Moreover, the two techniques stack very well, trimming almost 90% of the original byte count. With smaller scripts, results probably won’t be as good, but it is safe to assume an average reduction of an 80% in size, once both steps are applied.

**Concatenation and spriting**

To go further in the road of reducing HTTP requests and overhead, we may consider resource concatenation. Scripts and stylesheets are executed sequentially as they appear in the HTML, so it is possible to combine them into less files.

If our web application is very dependant on javascript, it is possible that our pages use many script resources, be it external libraries or internal. As developers, it may be useful to develop our own libraries and functions in different files, since it makes the code more reusable and easier to refactor, but that does not mesh well with performance. Since it is especially important to deliver stylesheets and scripts for a fast rendering time (as we will see in the next chapter).

Deciding which scripts should be stitched together can be done with a dependency map. If we know, for every view on the webapp, which scripts are used, we can concatenate those that only used together in that view, while keeping scripts used in other bundles separate, for better cache usage.

The case of stylesheets is easier. Since the additions between different site views should be minor, and CSS retrieving is very important for progressive loading, it is better to have only one css file, even if some parts of the CSS are not used at first. This way, we can cache it for every other view right away.

Spriting consists in stitching images together, and use CSS to display the desired part of this new image sprite. Stitching images together is a very good option for pages that use a lot of images in combination with their CSS to increase the visual attractiveness of the site without having to rely on lesser supported CSS options. Those image resources may be very abundant and very small, so they would delay the page load (loads of small resources will be capped by RTT). Services like SpriteMe (http://spriteme.org/) detect the spriting opportunities of the page and offer a fairly automated way to generate and integrate those sprites on any website.

<table>
<thead>
<tr>
<th>Compression</th>
<th>Size</th>
<th>Savings</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAW</td>
<td>274.080 bytes</td>
<td>0%</td>
</tr>
<tr>
<td>minified</td>
<td>93.064 bytes</td>
<td>66%</td>
</tr>
<tr>
<td>gzipped</td>
<td>81.352 bytes</td>
<td>70%</td>
</tr>
<tr>
<td>minified + gzipped</td>
<td>32.812 bytes</td>
<td>88%</td>
</tr>
</tbody>
</table>
CHAPTER 4. APPLICATION LAYER

Resource inlining

In the case of dealing with small resources, like small images or scripts, we might even consider embedding them in the HTML code itself. This is best saved only for small, rather static, content.

To inline binary resource, the src attribute may be a base64 representation of the object, instead of an uri. The format is as follows: 
\[
data: [mediatype] ;base64 , data
\]

```html
<img src="data:image/gif;base64,R0lGODlhAQABAIAAAAAAA
AAAAACH5BAAAAAAAALAAAAAABAEAAAAICTAEAOw==" alt="1x1 transparent (GIF) pixel" />
```

Since base64 encoding adds a 33% byte overhead, and some browsers even have hard limits on the size those inline resources can reach, it is better to keep them under 1 or 2 kB.

In general, those 4 points are considered good criteria to decide for or against inlining:

- If the files are small, and limited to specific pages, consider inlining.
- If the small files are frequently reused across pages, consider bundling.
- If the small files have high update frequency, keep them separate.
- Minimize the protocol overhead by reducing the size of HTTP cookies.

HTTP 1.1 performance rules

From what we have discussed until this point, the following rules will serve as guideline to improve our HTTP 1.1 applications:

1. **Use fewer HTTP requests.** Combine resources, inline them, use sprites or simply delete them. The less requests to perform, the faster it will load.
2. **Use a CDN.** Static content should be hosted as close to the client as possible. A CDN is a good way to distribute content and user perceived latency.
3. **Configure caching.** Add caching headers for all your resources. Use long cache deltas (like +10 years) and change the resource file name if a new version has to be deployed (to avoid stale cache issues).
4. **Compress the payload.** Use gzip for compressing text. Change the format or optimize the compression of your multimedia resources and minify both javascript and CSS.
5. **Avoid redirects.** Redirects add an additional RTT per request, so they should be avoided at all costs, especially for key resources.
Headers and cookies

HTTP does not compress headers, so they may represent a lot of overhead in a request. Moreover, since HTTP is stateless, information like cookies must be transmitted on every request, which is bad. A way around this is to use another subdomain for static content. Requests to a new subdomain won’t carry cookie data.

If we are using AJAX, with a lot of short json responses that don’t need authentication this may be a very good approach to save some bytes too, since in those kind of short and frequent responses the protocol overhead is around 80%, and burdening it with cookies will only make it worse.

HTTP 2.0 in detail

Now that we have come this far, it is rather clear that many of the workarounds needed to decrease the latency of existing HTML 1.1 deployments are rather inconvenient to deploy. Sharding, concatenation, resource inlining or spriting are techniques that seek better usage of the underlying TCP socket or faster downloads through parallelism. The latest HTTP 2.0 draft (Belshe et al. [2013]) provides an answer to most of the HTTP 1.1 woes by providing a new messaging frame, while maintaining the semantics of the protocol.

HTTP 2.0 won’t change the semantics of the protocol, only it’s framing. This will allow our existing applications to take advantage of the HTTP 2.0 performance boost (50% reduction in page load time) (Belshe and Peon) as soon as our clients and servers are HTTP 2.0 compatible.

On this chapter I will explain HTTP 2.0 based on the latest available draft (Belshe et al. [2013]). This version is very similar to the SPDyv3 implementation, since it served as the core of the new specification, but I expect that this content will be soon outdated by new drafts. Still, the core ideas will remain, and they are interesting enough to be introduced.

Overview

One of the main bottlenecks of HTTP 1.1 is the reliance in multiple connections for concurrent downloads. HTTP 1.1 had pipelining mechanisms that have been impossible to deploy due to intermediary interferences. Even if those middle boxes were a non-issue, pipelining still would suffer of head-of-line blocking. To prevent this and other problems, HTTP 2.0 adds a framing layer that allows multiplexing of streams over the same TCP connection.

The main 4 improvements over HTTP 1.1 are:
Multiplexed requests A single HTTP connection can handle any number of concurrent requests.

Prioritized requests Clients can request certain resources to be delivered first. This would allow critical resources to be delivered first without congestion problems caused by other, less important, requests.

Compressed headers Since HTTP headers send a significant amount of redundant data, compressing it will greatly reduce the overhead of the protocol.

Server pushed streams Servers may deliver content to clients without an explicit request on their behalf.

Layer abstraction

The new framing mechanism makes it necessary to introduce the following terminology in order to fully describe the data exchange in the new protocol.

Session Synonym for connection. Consists of a single TCP connection.
Stream A bi-directional stream within a session. May be a typical request-response pair or a push stream.
Message Sequence of frames that compound a logical HTTP message.
Frame Smallest unit of communication. May contain control data or payload. Each frame has an identifier to associate itself to a stream.

Figure 4.8: HTTP 2.0 layer abstraction, from Grigorik [2013c], ch. 12
Binary framing

Instead of sending HTTP headers delimited by CRLF characters, HTTP 2.0 uses a well defined binary frame, similar to a TCP segment or an IP packet.

Frame header

```
| Length (16) | Type (8) | Flags (8) |
+------------+----------+-----------+
| Stream Identifier (31) |
+-------------------------+
| Frame Data (0...) ... |
```

**Length** 16bit unsigned int indicating the length of the frame data.

**Type** 8bit type of the frame. Determines how the frame will be interpreted.

- Right now, there are 9 frame types defined.

- **Flags** 8 bits reserved for frame related boolean flags.

- **R** 1 bit reserved for future iterations of the protocol.

- **Stream identifier** 31bit identifier that determines which stream is related to the frame. Session wide messages have the 0 identifier.

**Frame data** Dependant on the frame type.

**Frame types**

- **DATA** Used to send payload. Each request or response may be sent using multiple DATA frames.

- **HEADERS + PRIORITY** Allows the sender to communicate the priority of the stream to the receiver and the headers of the request. Requests by the client will be sent using HEADERS + PRIORITY frames, while responses will be sent with a HEADERS frame.

- **RST_STREAM** Allows termination of a stream.

- **SETTINGS** Conveys configuration parameters related to the session as a whole. Must be sent at the start of a connection.

- **PUSH_PROMISE** Notifies an endpoint in advance that a certain resource will be sent in response to a certain request.

- **PING** Built-in mechanism to determine the RTT between endpoints.

- **GOAWAY** Gracefully kill a session.

- **HEADERS** Provide any number of header fields for a stream.

- **WINDOW_UPDATE** Built-in mechanism to regulate a session’s flow control.
Multiplexing

Since each frame contains information regarding the stream it belongs to, and TCP ensures in-order delivery of the packages it is possible to multiplex multiple requests on the same TCP stream. This is the single most important feature of HTTP 2.0, and the most critical one performance wise.

True multiplexing of streams enables:

- parallel requests and responses without HTTP head-of-line blocking.
- optimal TCP socket usage. HTTP sessions are now a mixed stream of data instead of bursts of requests and responses.

![HTTP 2.0 multiplexing of streams](image)

The fact that all the requests can be transferred over the same connection makes most of the HTTP 1.1 performance strategies obsolete. We no longer need to distribute the content over multiple domains or reduce the number of resources by inlining them.

Lowering the number of required TCP connections per server, makes it possible to keep them open for longer periods of time (the spec declares that only the server should close the connection, and that it should remain open for as long as possible.). From Belshe et al. [2013], section 3.1:

HTTP/2.0 connections are **persistent**. That is, for best performance, it is expected a clients will not close connections until it is determined that no further communication with a server is necessary (for example, when a user navigates away from a particular web page), or until the server closes the connection. Servers are encouraged to maintain open connections for as long as possible, but are permitted to terminate idle connections if necessary. When either endpoint chooses to close the transport-level TCP connection, the terminating endpoint MUST first send a GOAWAY (Section 3.8.7) frame so that both endpoints can reliably determine whether previously sent frames have been processed and gracefully complete or terminate any necessary remaining tasks.
Prioritization

With multiplexing, we have much more data to stuff into the TCP stream, but if we do so arbitrarily it may end up being counterproductive. Bad frame scheduling may cause delays in the delivery of key resources and, to prevent that, HTTP 2.0 allows stream prioritization.

The client can establish the priority of a stream at the start of a request, by sending a HEADERS+PRIORITY frame. Priority is represented as an unsigned 31 bit integer. 0 represents highest priority, while $2^{31}-1$ the lowest one.

It should be noted that the draft does not specify any algorithm for dealing with priorities. This is left to the endpoint implementations.

Header compression

Name/Value pairs in HEADERS and HEADERS+PRIORITY frames are compressed, although the method is still being discussed. The SPDYv3 implementation uses zlib with a custom dictionary encoding (Yang et al.), but since a vulnerability that allowed session hijacking was discovered in late 2012 (see Hoffman) a replacement has yet to be found.

Server push

Push transactions enable HTTP 2.0 to send multiple replies to a client for a single request. This is, in a way, an evolution to the inlining mechanism in HTTP 1.1. Traditionally, the server sends the requestes HTML to the client and it is the client that must discover and request every linked resource in the page. This communication schema introduces delay, since most of the resources have to wait a full RTT before they can be requested. With push transactions, the resource discovery is left at the server side, allowing it to send resources that the client is likely to discover.

To prevent race conditions (resources to be pushed are discovered and requested by the client, before being sent) HTTP 2.0 introduces the PUSH_PROMISE frame, which announces to the client which resources will be pushed in response to his request. The client has to explicitly deny PUSH_PROMISE it does not want to receive by issuing a stream error.

The push transaction workflow is as follows:

1. Client creates a new stream and send a request to the server
2. Server sends PUSH_PROMISES for the related content, that are spawned in new streams. Those promises include the URI of the resources and validation headers to determine whether or not the client has the resource in cache.
3. Server sends requested resource.
4. Client receives PUSH_PROMISES:
   a. Do nothing. Accept the push.
   b. Issue CANCEL stream error. Deny the push. May be due to the resource being already cached or PUSH_PROMISE pointing to a cross site URI (security concern).
5. Server sends promised data. May cancel transaction if it receives a CANCEL error before sending the resource.

![HTTP 2.0 session diagram](image)

stream 1: /page.html (client request)
stream 2: /script.js (push promise)
stream 4: /style.css (push promise)

Figure 4.10: Single request instantiates multiple response streams, from Grigorik [2013c], ch. 12

**Deployment**

Even if client and server can speak HTTP 2.0, we need a method to convey this information to the other endpoint and upgrade the connection. Nowadays, the most reliable and efficient method to do so is to employ the *Application Layer Protocol Negotiation* (ALPN) implemented in the TLS layer.

Since network intermediaries may tamper unencrypted traffic, most new application protocols are deployed using a TLS layer. Encrypting the traffic obscures the data and the protocol at the same time, so it is the most convenient way to avoid interferences from the network environment.

ALPN is able to negotiate the protocol upgrade using the TLS handshake, so there is no additional RTT (in contrast to a standard HTTP upgrade mechanism). The negotiation process is as follows:

- The client appends a `ProtocolNameList` field with a list of supported protocols into the ClientHello message.
- The server responds with a ServerHello message, containing a `ProtocolNameList` with the selected protocol.
The server must select only one protocol. If the server does not support any of the protocols proposed by the client, the connection is closed.

![Figure 4.11: TLS handshake, from Grigorik [2013c], ch. 4](image)

**HTTP 2.0 performance rules**

Most of the performance recommendations for HTTP 1.1 are no longer needed in HTTP 2.0:

1. **Use fewer HTTP requests.** Since all requests are multiplexed and the headers compressed, more requests will have less of an impact in the overall performance of the webpage.
2. **Use a CDN.** A CDN is still a good way to cut on latency. Still, techniques like domain sharding are no longer necessary, since HTTP 2.0 allows multiplexing.
3. **Configure caching.** Caching is as important in HTTP 2.0 as it was in 1.1. In that regard, the same mechanisms apply to both protocols.
4. **Compress the payload.** 2.0 allows the same mechanisms to compress the payload, but also compresses the headers, making small requests (like AJAX updates calls) have much less overhead. CSS spriting and javascript concatenation is no longer necessary, since requests are much cheaper in comparison to 1.1.
5. **Avoid redirects.** This still applies. Redirects should be avoided, since they add an RTT per request.

HTTP 2.0 also gives us more tools that will decrease the latency of the protocol, like the ability to push resources to the clients. Other interesting characteristics, like making some headers valid session or stream wide, are also being discussed into the protocol. This would make HTTP 2.0 a stateful protocol and allow us to send information like cookies or user agents only once per connection.
CHAPTER 4. APPLICATION LAYER

Summary

As we have seen in this chapter, HTTP was originally designed to deliver static pages with few resources. Even though HTTP 1.1 is a major evolution of the original protocol, it got some key features wrong (pipelining) and has grown old for our current needs.

There are many workarounds around the limitations of the 1.1 protocol, but they encumber the development of web applications and don’t solve the fundamental problems behind the protocol.

HTTP 2.0 is the solution to most of 1.1 woes, adding true multiplexing, push and prioritization. In table 4.3 I have made a comparison between the key features of both protocols.

<table>
<thead>
<tr>
<th></th>
<th>HTTP 1.1</th>
<th>HTTP 2.0</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Transport</strong></td>
<td>Keep-alive</td>
<td>Keep-alive</td>
</tr>
<tr>
<td><strong>Concurrency</strong></td>
<td>Multiple connections</td>
<td>Multiplexing</td>
</tr>
<tr>
<td><strong>Caching</strong></td>
<td>Cache-control headers</td>
<td>Cache-control headers</td>
</tr>
<tr>
<td></td>
<td>Validation headers</td>
<td>Validation headers</td>
</tr>
<tr>
<td><strong>Compression</strong></td>
<td>Content</td>
<td>Content</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Headers</td>
</tr>
<tr>
<td><strong>Push</strong></td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td><strong>Prioritization</strong></td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Table 4.3: HTTP 1.1 and 2.0 comparison

Chapter references

This chapter is based on Grigorik [2013c], ch. 12. The caching section has been expanded using Gourley et al. [2002], ch. 7 and Souders [2007] ch. 13. For a more complete version of the HTTP 1.1 performance rules I recommend Souders [2007] and Souders [2009]. The HTTP 2.0 section is basically a summary of the current specification draft (Belshe et al. [2013]).
Chapter 5

Presentation Layer

To achieve the best perceived performance from the client, the rendering of the page should be progressive. To aid the browser in this matter, it is important to understand how it constructs the parse tree, in order to deliver key resources as soon as possible.

![Browser processing pipeline](image)

Figure 5.1: Browser processing pipeline, from Grigorik [2013c], ch. 10

The browser processing pipeline builds the Document Object Model (DOM) out of the HTML file and a lesser known CSS Object Model (CSSOM). Once we have those objects models, we can build a rendering tree and start painting the webpage.

This seemingly simple schema starts to get confusing when javascript is introduced. Javascript can modify both the DOM and the CSSOM, which complicates the dependency graph and may cause quite a performance hit in our page load times.

We have to be especially aware of `document.write` calls, since they are synchronous and block DOM parsing and construction. We also have to be aware of javascript calls that query the CSSOM. If we perform one of those, the query can’t be answered until the CSSOM is available, and while the query is waiting, so it is the construction of the DOM.
Since the CSSOM is so important, we have to deliver CSS resources as soon as possible. To ensure that this will be the default scheduling behaviour of the browser, it is strongly recommended to put the stylesheets at the top of the HTML page.

Scripts block the rendering of the content placed below themselves until they have finished running, so it is strongly recommended to put scripts at the bottom of the HTML page.

3rd party snippets

A lot of websites use 3rd party scripts to add features like analytics, twitter or facebook integration, etc. Those scripts are typically retrieved from the 3rd party site each time the page loads, since it makes it easier for that 3rd party to deploy updates seamlessly. The fact that we are retrieving a resource from this 3rd party URL might introduce a single point of failure in our webpage, if we are not careful.

As we have said before, scripts block all elements below themselves, so if the snippet loads synchronously and the service is overloaded or unavailable, our page may take a very long time to load. Moreover, if this script is placed on the top, we will be staring at a blank page until a response is received or the connection times out (which can take as long as 120 seconds on Internet Explorer).

Now that we are aware that 3rd party snippets should be loaded asynchronously, we have to be able to distinguish between async or synchronous scripts. Here are 2 examples:

```html
<script type="text/javascript" src="https://platform.twitter.com/anywhere.js"></script>
```

This is a synchronous script. Basically, any script with a src attribute will freeze until the resource pointed by the uri has been downloaded.

```html
<script type="text/javascript">
var _gaq = _gaq || [];
_gaq.push(['_setAccount', 'UA-XXXXX-X']);
_gaq.push(['_trackPageview']);

(function() {
    var ga = document.createElement('script'); ga.type = 'text/javascript'; ga.async = true;
```
var s = document.getElementsByTagName('script')[0]; s.parentNode.insertBefore(ga, s);()}
</script>

This Google Analytics script is loaded asynchronously. It creates a `<script>` element, pointing to the 3rd party script, sets the async property to true and injects the new DOM object in the webpage.

The async property used was introduced in HTML5 and it basically tells the browser to execute the script contents as soon as they are available without blocking.

To transform a synchronous snippet to async, we can simply add the async keyword like this:

```html
<script async type="text/javascript" src="https://platform.twitter.com/anywhere.js"></script>
```

This async attribute is especially useful to prevent external issues to bring our page down. We can also use it for our own scripts, although we’ll have to be careful: async will trigger the script as soon as it is available. That means that if the script depends on a library that has yet to be loaded, its execution will fail. To allow asynchronous script execution, while maintaining the order in which they are executed, the defer keyword was also added. defer loads scripts asynchronously but guarantees that they will be executed in the order they occur in the page. More information in that regard can be found on Gentilcore [2010].

**Summary**

In this chapter we remark how a careless positioning of the stylesheets and javascript elements in the HTML may increase the rendering time of the webpage.

Even though modern browsers do much more to decrease webpage load times, like prefetching the DNS of our most visited websites, or automatically initiating a handshake with our top ranked pages on startup, the tricks mentioned in this chapter are the most relevant from the perspective of a web application developer.

**Chapter references**

The content on this chapter is based on Grigorik [2013d] ch. 10, and Souders [2007] ch. 5 and 6. The 3rd party snippets section is based on a presentation by Steve Souders at Fluent Conference 2012 (Souders [2012]).
Chapter 6

Performance Assessment on LdShake

To put an end to this project we will do a performance assessment on LdShake (http://ldshake.upf.edu) (Hernández-Leo et al. [2011]). LdShake is a web application to share and edit learning design solutions mainly tailored at teachers. LdShake is built on top of the PHP Elgg framework (http://elgg.org/), which makes development of social networks easier. It uses a fair amount of javascript, which is mainly used to add interactivity to the webpage, and elements like text editors. Once loaded, though, the pages don’t make use of AJAX to modify the content dynamically. The platform is also quite sparse in image resources, so each page weights much less than the typical 1.5MB we saw in the introductory chapter.

Overall, we could say that LdShake is lighter than the typical website, has less resources (around 40 per page instead of a 100) and does not load content dynamically (although the content generation is, indeed, dynamic). Despite these differences, I still believe that LdShake performance analysis can be easily extrapolated to perform similar assessments on other websites.

Setting up the testing environment

First of all, we have to prepare our testing environment. I have created a VM running Ubuntu Server 12.04 (same OS as production system) and configured following the instructions that appear the ldshake tarball (http://sourceforge.net/projects/ldshake/).

Once the VM is up and running, we will also have to add a traffic shaper to replicate the network conditions our users are most likely to have. To do this,
Figure 6.1: LdShake running from the local VM
I will use OSX Network Link Conditioner. Network Link Conditioner is an optional developer tool in Xcode that artificially alters the link capabilities of our networking connections. It can be installed by opening XCode and accessing XCode > Open Developer Tool > More developer tools.

![Network Link Conditioner with the test settings](image)

Now that we have the traffic shaper installed, we have to set up the network conditions for our tests. To do so, I have looked up a 2012 study on the average Spanish broadband connections (http://www.testvelocidad.es/estudio-velocidad-2012/). It basically states that a typical user has a 7.1Mbps downlink and a 435Kbps uplink. I have also added a 60ms latency, which is on par with the FCC study numbers on the network layer chapter and a package loss of 1%, which is also a typical value in a wired connection.

If we ping the local VM from our traffic shaped machine, we can certify that everything is working as expected:

```
$ ping -c 1 ldshake.local
PING ldshake.local (172.16.246.134): 56 data bytes
64 bytes from 172.16.246.134: icmp_seq=0 ttl=64 time=63.399 ms

--- ldshake.local ping statistics ---
1 packets transmitted, 1 packets received, 0.0% packet loss
round-trip min/avg/max/stddev = 63.399/63.399/63.399/0.000 ms
```
LdShake architectural overview

LdShake uses the MVC pattern. This means that presentation data, like html templates, scripts, stylesheets and images should be kept apart from the business logic, which makes front-end changes easier to deploy without affecting the back-end.

Initial assessment

Since LdShake uses templates to construct each view and shares a lot of resources between them, we will only do performance assessment on a subset of views, which will be:

- login page
- first steps page
- create ldshake (new content) page
- browse available ldshakes (content) page

Using the Chrome developer tools and the PageSpeed Insights extension (https://developers.google.com/speed/pagespeed/insights) we will run an analysis on those pages and detect the problems they may have. The given punctuation is a custom index created by Google that gives us a good idea of how optimized a webpage is (from a front-end point of view). Those indexes were first introduced with the Yahoo UI tools and correlate greatly with decreasing page load times, so they serve as a good measure to quantify the front-end optimization of a webpage.

Login view

Score: 69 / 100

Description: The login view is a very simple html page, with some inlined javascript for the google analytics and some images.

Problems:

1. There are no cache headers. Images don’t have any cache information.
2. Usage of redirects. Some images are misnamed (.PNG instead of .png). This triggers a costly 301 Moved Permanently redirect.
3. Images could use better compression (-28% size).
4. Images could be scaled down (-65% size).
Figure 6.3: LdShake views to be tested. From left to right: login, first steps, create content, browse content
First steps view

Score: 72 / 100

Description: The first steps view is an introductory page to the site, showing some functionality and the necessary extensions to be installed to get a full experience.

Problems:

1. There are no cache headers. Images don’t have any cache information. Stylesheets or javascript don’t have any cache information either.
2. Usage of redirects. Some images are misnamed (.PNG instead of .png).
3. CSS sprites could be employed.
4. Images could use better compression (-28% size).
5. Images could be scaled down (-65% size).
6. Javascript and CSS are not minified.
7. Some javascript resources are very small and should be inlined.

Browse view

Score: 78 / 100

Description: The browse view displays a list with all the content created by LdShake users.

Problems: Same as previous view.

New LdShake view

Score: 83 / 100

Description: Rich text javascript editor.

Problems: Same as previous view. The mark improves because it has less unoptimized images.

Performance optimizations

I have performed 2 rounds of optimizations. In the first one I have applied most of the HTTP 1.1 optimizations techniques discussed in the 4th chapter. In the second one, apart from the HTTP 1.1 optimizations, I have enabled SPDYv3 as the application protocol (instead of using HTTP).

To solve most of the issues pointed by the initial assessment I have performed the following changes in LdShake:
- Enable mod_expires in apache.
- Concatenate commonly used javascript files.
- Serve static css instead of dynamic. Delete parameter to allow hashing.
- Minify all javascript and CSS files.
- Optimize images for better compression
- Rename images to avoid 301 redirects.

The default .htaccess provided by elgg already configured caching and Etags. By default, files are cached with a 10 year expiration date. That basically forces us to deliver any kind of update with a different filename, but as we will see, caching does wonders to improve the performance of the website. It is worth it.

The default template used by all views loaded jQuery, jQuery-ui and another pair of custom scripts every time. I have joined them and minified it, so this should also help.

The rest of the steps consists on applying UglifyJS to the javascript files of the project and YUICompressor to the CSS. I have also recompressed the website images and added some CSS sprites where they could be applied. To generate the sprites and the modified CSS I have used the spriteme bookmarklet (http://spriteme.org/), created by Steve Souders.

With all this optimizations performed, it is time to see how the new version scores:

<table>
<thead>
<tr>
<th>View</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>Login view</td>
<td>90 / 100</td>
</tr>
<tr>
<td>First steps view</td>
<td>96 / 100</td>
</tr>
<tr>
<td>Browse view</td>
<td>99 / 100</td>
</tr>
<tr>
<td>New LdShake view</td>
<td>98 / 100</td>
</tr>
</tbody>
</table>

Table 6.1: PageSpeed scored after the optimizations

In a last attempt to improve performance even further, I have decided to configure mod_spdy and test the webapp over SPDY, to see how much of an improvement in brings over HTTP. To install mod_spdy on apache, I have downloaded the packages from the project site (https://code.google.com/p/mod-spdy/) and added the following to the apache config file (/etc/apache2/apache2.conf):

```html
<IfModule spdy_module>
    SpdyEnabled on
    SpdyDebugUseSpdyForNonSslConnections 2
</IfModule>
```
With this, we are basically telling the server to enable SPDY and serve it over unencrypted connections by default. This means that the client must connect using SPDY directly, since the connection won’t upgrade from HTTP to SPDY automatically. We can force Chrome into a SPDY only mode by running it with the following parameters:

```
--use-spdy=no-ssl
```

Figure 6.4: LdShake on SPDY. Session as captured by Chrome net-internals

I am aware that SPDY would usually be employed over HTTPS, using ALPN to upgrade the connection to SPDY, but for testing purposes, I wanted to see how SPDY performed raw.

**Measuring**

Apart from this performance index given by tools like PageSpeed Insights or YSlow (http://developer.yahoo.com/yslow/), I wanted to have real numbers to quantify the speed increase between versions. In order to do so, I have performed the following experiment:
MEASURING

• For each view:
  – Reload the page 10 times without cache. Record onLoad time.
  – Reload the page 10 times from cache. Record onLoad time.

This test has been done on all 4 views and for all 3 versions of the site:

• Original site. HTTP 1.1
• Optimized site. HTTP 1.1
• Optimized site. SPDY

I have extracted the average and median onLoad times per case, and I also have recorded the number of requests and transfer size per load. All the data can be accessed from http://goo.gl/HibSN. This url points to a Google Docs spreadsheet with the complete version of the experiment.

<table>
<thead>
<tr>
<th>View</th>
<th>B. F. PLT</th>
<th>B. C. PLT</th>
<th>O. F. PLT</th>
<th>O. C. PLT</th>
<th>SPDY F. PLT</th>
<th>SPDY C. PLT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Login</td>
<td>771ms</td>
<td>606ms</td>
<td>715ms (-7.2%)</td>
<td>282ms (-54%)</td>
<td>498ms (-35%)</td>
<td>262ms (-57%)</td>
</tr>
<tr>
<td>First Steps</td>
<td>1205ms</td>
<td>944ms</td>
<td>912ms (-24%)</td>
<td>236ms (-75%)</td>
<td>799ms (-33%)</td>
<td>244ms (-74%)</td>
</tr>
<tr>
<td>Browse</td>
<td>1165ms</td>
<td>870ms</td>
<td>958ms (-18%)</td>
<td>228ms (-74%)</td>
<td>755ms (-35%)</td>
<td>240ms (-72%)</td>
</tr>
<tr>
<td>New content</td>
<td>2095ms</td>
<td>1100ms</td>
<td>1250ms (-40%)</td>
<td>382ms (-65%)</td>
<td>1495ms (-29%)</td>
<td>417ms (-62%)</td>
</tr>
</tbody>
</table>

Table 6.2: Median page load times per view and optimization. Percentage indicates decrease in PLT against baseline.

We have obtained a substantial improvement in prime page load time in both cases. In this test, since there is more data to download, SPDY can shine with the use of multiplexing. I would expect that bigger websites with many more resources would increase the SPDY lead by a larger margin.

The improvement in cache PLT is spectacular. It really shows how much can cache improve the user experience. In this case, only the html had to be downloaded from the server. Since all the other resources has already been fetched and stored, they loaded instantly. In this case, since the connection only has to download an small HTML file, SPDY does not show any real performance improvement with standard HTTP 1.1. In fact, it performs a little bit worse.

<table>
<thead>
<tr>
<th>View</th>
<th>B. Fresh tx sz</th>
<th>B. Cached tx sz</th>
<th>O. Fresh tx sz</th>
<th>O. Cached tx sz</th>
</tr>
</thead>
<tbody>
<tr>
<td>Login</td>
<td>115kB</td>
<td>7,3kB</td>
<td>96.2kB (-16%)</td>
<td>3,3kB (-55%)</td>
</tr>
<tr>
<td>First Steps</td>
<td>228kB</td>
<td>27,5kB</td>
<td>202kB (-11%)</td>
<td>3,1kB (-89%)</td>
</tr>
</tbody>
</table>
CHAPTER 6. PERFORMANCE ASSESSMENT ON LDSHAKE

<table>
<thead>
<tr>
<th>View</th>
<th>B. Fresh req</th>
<th>B. Cached req</th>
<th>O. Fresh req</th>
<th>O. Cached req</th>
</tr>
</thead>
<tbody>
<tr>
<td>Login</td>
<td>13</td>
<td>13</td>
<td>12</td>
<td>12</td>
</tr>
<tr>
<td>First Steps</td>
<td>35</td>
<td>35</td>
<td>27</td>
<td>23</td>
</tr>
<tr>
<td>Browse</td>
<td>25</td>
<td>25</td>
<td>22</td>
<td>16</td>
</tr>
<tr>
<td>New content</td>
<td>39</td>
<td>39</td>
<td>34</td>
<td>26</td>
</tr>
</tbody>
</table>

Table 6.3: Transfer size per view. Percentage indicates decrease in transfer size against baseline

Table 6.4: Number of requests per view.

From the transfer size and requests tables we can also see the decrease in size and number of requests. This is mainly thanks to the better compression (minification and better image encoding) and also thanks to the spriting of images in the CSS and the concatenation of javascript files. I have not included the transfer sizes for SPDY since the Chrome Dev Tools does not report them. The number of requests for SPDY is the same than those for the optimized HTTP site.

In the last four figures, we can appreciate the variance of the results in each experiment. As we can see, the original version of each view has higher variances and higher page load times, while the optimized versions all show lower variances and lower page load times, especially the cached view.

The variance can be explained mostly due to our package drop configuration in the traffic shaper (1% out, 1% in). The optimized versions show lower variances since the transferred bytes and requests per view has decreased due to the performed optimizations.

Summary

In this chapter we have finally put to use the attained knowledge in the previous ones. We have seen that it is indeed possible to improve the delivery performance of our websites without changing much of our service, and it has also become apparent that HTTP 2.0 performs better than 1.1, but not noticeably better for lighter pages such as LdShake views. It would be interesting to test HTTP
Figure 6.5: Sample variance for the login view test

Figure 6.6: Sample variance for the first steps view test
Figure 6.7: Sample variance for the browse view test

Figure 6.8: Sample variance for the new content view test
2.0 on pages filled with multimedia content, like newspapers or flickr. It would also be interesting to make use of its new push capabilities and see how they compare with the typical request-response model.
Conclusions

In this project we have found that latency is the number one cause of slow page load times. It has also been shown how latency can’t be reduced from the physical layer, so it’s an issue that must be tackled higher in the network stack.

The current network stack has not been built to enable low latency and realtime services, which is where web applications are heading. To solve this dissonance between what we want and what we have, there are a lot of workarounds and fixes in the TCP and HTTP 1.1 protocols. Still, those fixes do not solve the fundamental problems, which will be better resolved in the future version of HTTP 2.0.

Finally, we have performed a performance assessment on a current web application, LdShake. In this performance assessment, we have demonstrated that the acquired knowledge in the previous section is indeed useful, and we have also shown that front-end optimizations can yield an enormous decrease in page load times (as much as 75% in our tests), which will hopefully translate to more successful services.
Ending thoughts and future research

The typical attitude towards front-end optimizations is that it is out of the technician hands and it should not be bothered with. Although it is true that there are many factors out of our hands when delivering data to our clients, it is no less certain than doing our best in that regards pays off big time. As we have seen in our performance assessment, front-end optimizations go from correctly configuring our webserver modules to less than elegant css spritings. There are a lot of aspects that impact the perceived user experience, some of which are easier than others to deploy, but most certainly much easier to implement than performing changes in the underlying architecture of our service.

Now that we have demonstrated the value of this kind of performance assessment on our web applications, it is important to remark that most of those optimizations are no more than workarounds. While those hacks enable our present applications, as we push the web further, and mobile takes over (and it soon will) we’ll find us in need for a higher performance, lower latency protocol stack.

Those new protocols have been recently flourishing, with the creation of SPDY (soon to be HTTP 2.0) or other like-minded protocols such as WebSockets. In fact, so much is changing in the protocol stack, that even new transport layer protocols are emerging (see Lardinois [2013]).

For future research on the subject, I would find especially interesting to investigate how this protocols could be employed to improve out mobile experiences. Approaches like Amazon Silk, were the traffic is routed through a single SPDY connection to the cloud, and then routed to the internet, should be explored further. If those new protocols deliver on their promises, they could be used to dramatically improve the network connectivity of millions of users around the world, especially in emerging economies. And whomever taps into this market first might as well have taken the 21st century’s crude field for themselves.
CHAPTER 6. PERFORMANCE ASSESSMENT ON LDSHAKE
Bibliography


Nagle, J. Congestion Control in IP/TCP Internetworks. .


Rhee, I. and Xu, L. CUBIC: A New TCP-Friendly High-Speed TCP Variant.


Yang, F., Amer, P., Leighton, J., and Belshe, M. A Methodology to Derive SPDY’s Initial Dictionary for Zlib Compression.