University of Bergen

Faculty of Social Sciences

Department of Information Science and Media Studies

Master Thesis

Performance as Design

Techniques for Making Websites More Responsive

Author: Eivind Mjelde *Supervisor: Andreas Opdahl*

June 4, 2014

Abstract

Websites today have to cater to a fragmented device ecosystem. Previous web design trends utilizing fixed-width layouts, aimed at desktop computer browsing, have been replaced by new approaches embracing fluidity, accessibility and context-awareness. Responsive web design is one such approach, merging desktop and mobile layouts into a single codebase. With this merger one feature is often overlooked, namely performance.

In this thesis I have aimed to identify inherent performance challenges within the responsive web design paradigm. I have looked at how existing performance optimization techniques can be utilized to alleviate these challenges by comparing their effects under both mobile and desktop browsing contexts.

The methods used included collection and discussion of the advantages and disadvantages of existing performance optimization techniques, as well as exploratory research through experiments. The aim of experimentation was to observe how different techniques work in a controlled setting in order to identify how performance was improved, and at which cost.

The findings show that several existing techniques can be applied to improve web performance, some especially targeting responsive design and mobile use cases. However, the problem with many of these techniques is that they are workarounds for deficiencies inherent in current web protocols, markup technologies and browsers. While these workarounds can be proven to enhance web performance in certain use cases, most also possess inherent disadvantages. This emphasizes the demand for new technologies. HTTP 2.0 as well as emerging responsive image technologies are predicted as potential solutions to some of these deficiencies.

Preface

This thesis concludes my Master in Information Science education at The University in Bergen (UiB). The thesis was carried out over two semesters, fall 2013 to spring 2014, at the Department of Information Science and Media Studies.

I would like to express my appreciation of the people who have helped during the course of my studies. First and foremost, I would like to thank my supervisor Andreas Opdahl for his guidance, feedback and optimism during the ideation and writing phases.

I would also like thank my fellow students, friends, family, and especially my parents, for continued support and encouragement.

A special thanks goes out to the guys in room 602 (Yngve, Per-Øyvin, Ola, and Øyvind) for input and discussions on academic work and everything else under the sun, and for good camaraderie in general.

Last, but not least, I would like to sincerely thank my girlfriend Ida. Without her continued love and support this process would have been that much harder. Thank you for putting up with my absent-mindedness in the final weeks of writing, and for always being a source of inspiration.

Contents

List of Figures

List of Tables

Listings

1 Introduction

The web today is something quite different from what existed just a few years ago. We do not have to travel far back in time to see applications, websites and web services that where created almost unanimously with the desktop[1] computer in mind. This has changed radically over the last decade, as more and more Internet-connectable devices have been introduced to both the consumer and the professional market. Everything ranging from smart phones and tablets, to desktop computers and television sets, are today continuously connected to the Internet. This development has happened in tandem with increasing hardware and browser capabilities. The biggest change with regards to how we browse the web has come with the development and proliferation of the modern smartphone.

The earliest Internet-connectable mobile phones had very limited capabilities when accessing web content and running native applications. But with the advent of the smartphone, utilizing advanced operating systems (OS) such as iOS and Android, and with hardware that allowed running more complex and feature-rich applications, the differences between mobile and desktop computing got smaller at an increasing speed. Mobile web browsers, like the ones native to iOS and Android, today share very similar browser engines to those found in their desktop equivalents. This gives smartphones the ability to utilize advanced features when structuring and visualizing web content, features built upon web standards such as HTML5, CSS3 and various client-side scripting languages (most commonly JavaScript).

As mentioned, the development of smartphones and tablets has radically changed the way we consume web content. Statistics show that more and more people are using mobile clients to access the web than before (Cisco, 2013). The statistics show that our usage patterns on the web have changed, and with it the need for a complimentary presentation of content and services has emerged.

[1]The term desktop in this thesis is also used to denote portable laptop computers.

Until recent years the norm for web page design was based on rigid structures with fixed-width layouts. These layouts were made to conform to average desktop monitor resolutions. On occasion, as computer display technology progressed, developers would scale up the fixed width of their designs in accordance with increases in average screen resolutions. Mobile devices often have lower resolutions and smaller physical screen sizes than that of an average desktop monitor or laptop. This causes problems when mobile device users try to access content built on a rigid layout structure targeting larger screens. A common problem is when designs are based on a fixed screen width that is larger than what a mobile device can offer. In many cases this forces the user to scroll both vertically and horizontally in order to access all content on a page. This can make it tedious and disorganizing to navigate, and offers the user a poor experience of the website. Another source of problems is the fact that many websites are built for interaction through use of a mouse and keyboard. Buttons, menus, forms and other elements are therefore rarely suitable for the touch interfaces that are most common on mobile devices. The challenge arising is how to optimize availability and presentation of web content across all possible combinations of devices (phones, laptops, desktop computers, game consoles, etc.) and browsers.

An early solution to the problem consisted of creating multiple versions of the same web site, one targeting small screen sizes, another targeting large screen sizes. The versions for smaller screens would often be delegated to mobile subdomains (also labeled M Dot websites (Fehrenbacher, 2007)), while versions for larger screens would use the default domain. Browser sniffing[2] or JavaScript could then be used to identify which browsers or screens sizes were used when visiting a site, and handle redirecting the user to the appropriate domain. This approach let developers tailor each version of a website to different devices. The problem with creating separate domains is that content needs to be duplicated across domains, which increases maintenance complexity by having to keep several systems up to date simultaneously. The disadvantages of creating separate sites for mobile and desktop contexts led to the emergence of new design approaches

[2]A technique to identify OS/browser by checking the User-Agent string on initial request.

such as responsive web design (henceforth labeled RWD). Leveraging adaptive content presentation, offered by CSS3 media queries, as well as fluid layouts and images, RWD allowed developers combine mobile and desktop websites into a single code base. The concept of RWD is more fully explained in section 2.1.

Early debates surrounding RWD focused on challenges of how to best handle content layout and user interaction (UX). Common topics were how to properly order and scale content for different screen sizes, and how to ensure usability for elements like navigation, forms and tabular data in mobile contexts. Much has happened since the early days of RWD. Newer style rules allow us handle layouts more flexibly (such as the CSS3 Flexible Box Layout Module), and front-end frameworks like Bootstrap[3] and Foundation[4] has made it easier to quickly develop standards compliant responsive websites with broad browser support. Still, there is one inherent challenge of RWD which is often overlooked, that challenge pertains to web performance.

What can be considered both a feature and a drawback for RWD is the fact that the same content and resources can be loaded for all screen resolutions. The positive aspect of this is that content is available to users regardless of which device they are using to access it. However, having exactly the same content, and loading the same resources, for both mobile and desktop versions of a web page is not always desirable. Therefore many web developers started using media queries to hide content on smaller screens. This made it easier to handle presentation of content on smaller screens, but did not hinder the web browser from downloading the same resources regardless of being visible or not.

Utilizing the same resources is particularly detrimental with regards to images and graphics, considering that their file sizes are usually much larger than textual resources. Graphics used for styling purposes are often hidden in mobile layouts, while content images are scaled down from their original size to fit smaller screens. Downloading the same resources for all devices therefore wastes bandwidth which could be saved if devices with smaller screens were served downsized images. The problem of wasted bandwidth is am-

[3] http://getbootstrap.com/

[4] http://foundation.zurb.com/

plified by the fact that mobile devices usually have less hardware capabilities, and often operate on slower network connections than desktop computers. Less powerful CPUs, graphics processing and JavaScript engines mean that mobile devices will spend more time rendering web pages, and smaller browser caches mean that resources are only stored locally for shorter durations of time. In addition, mobile devices frequently use mobile networks to connect to the Internet. Since mobile networks need to communicate with cell towers to propagate network traffic, a significant delay is added. Therefore, in order to ensure a good user experience for all devices, it is particularity important to optimize for performance on mobile devices when utilizing RWD. This process will serve as the core subject of this thesis.

In his much cited book *High Performance Web Sites*, Souders (2007, p. 5) introduces the concept of the "performance golden rule": "Only 10-20% of the end user response time is spent downloading the HTML document. The other 80-90% is spent downloading all the components in the page". This rule states that most of the time spent on downloading a web page happens on the front-end, that is, downloading the resources necessary to create the user interface. Back-end performance is fast because its processes happen internally on the server. Front-end resources on the other hand have to travel over the network from server to client. The delay imposed by communication over the network is much larger than that of the delay in a server response. How to most effectively download front-end components is therefore imperative in an effort to improve web page performance.

1.1 Research Question

As previously stated, RWD creates challenges for how to load the proper resources on mobile devices. This thesis therefore focuses on how we can make the downloading of resources more context-aware. A big aspect of this is connected to how we can selectively load appropriately resources for different devices. Since images and graphics often make up the largest part of a page's, weight we want to focus especially on how to to handle them in a more context-aware fashion.

This thesis wishes to examine how to make web pages more *device-agnostic* with regards to performance. That is, how to offer the best user experience for both mobile and desktop contexts by improving page loading times, render times, and reducing bandwidth consumption where possible.

To support this goal the following research questions have been formulated.

Research Questions:

1. Which techniques exist to improve front-end performance of device-agnostic websites?
2. What advantages and disadvantages must be considered when selecting and implementing these techniques?

1.2 Methodology

The purpose of this thesis is to examine several techniques that are claimed to improve web page performance, either in the form of reduced download sizes, reduced download times, faster render times, or a combination of two or more of these factors. We look at how the techniques affect performance both in mobile and desktop settings, in hope of identifying which are particularly beneficial for web pages utilizing responsive design. The thesis follows an essayistic format. As a result it does not rigorously apply any particular methodology throughout. Yet certain parts of the thesis use aspects of standard empirical research methodology. These are mainly the parts pertaining to performance testing. In this section, a brief summary will be given on how this thesis utilizes experiments as a mean for exploratory research, creating an adapted methodology that merges an essay-like structure interspersed with elements of scientific methodology.

It is important to note that the inclusion of the experimental section was done as an addition to theoretical discussion. Therefore experiments have not been executed as rigorously as scientific method might demand in order to provide statistically significant

evidence. But as stated by Tichy (1998, pp. 33-34), no amount of experimentation provides proof with absolute certainty. Still, we use it to test theories, and for exploration of new phenomena. The use of observation and experimentation is also said to be helpful in finding new, useful, and unexpected insights. In the case of this thesis, experiments are not conducted to directly prove a hypothesis, but rather as a way to probe the efficiency of these performance enhancement techniques and to fuel further discussion.

In the realm of web-related content design and front-end performance optimization, few structured academic research papers have been written. Therefore much of the theoretical foundation, and discussion of techniques, will be based on a comprehensive collection of white-papers, books, articles and blog posts written by organizations and people within the web development community. In doing so, this thesis attempts to accumulate the unstructured knowledge pertaining to the fields of interest into a more academic context.

Experiments as part of methodology is most commonly used within the natural sciences. The aim of experiments in the context of scientific methodology is to "find universal laws, patterns, and regularities" (Oates, 2005, p. 284). This is commonly done through experiments under controlled (laboratory) conditions. An experimental research strategy "investigates cause and effect relationships, seeking to prove or disprove a causal link between a factor and an observed outcome" (Oates, 2005, p. 127). This is done by formulating a hypothesis which is to be tested empirically through experiments. This thesis omits a single hypothesis in favor of using the research questions as a foundation for performing experiments on several techniques and technologies, in an effort to identify potential for web performance optimization.

Experiments are usually carried out in a three-step process:

1. The observation or measurement of a factor.
2. Manipulation of circumstances.
3. Re-observation or re-measurement of the factor from step 1 to identify any changes.

By following this process, the aim is to compare treatments, finding which factor(s) is

the cause, and which factor(s) is the effect. One should be able to predict or explain the causal link between two factors by means of the theory used to build the initial hypothesis. In order for the observed outcomes of the experiments to be considered proof of causal links, it is vital that the experiments are repeated many times under varying conditions. This is done to ascertain that observed or measured outcomes are not caused by unrecognized factors (such as equipment failure). Through this rigorous testing we are ensuring *repeatability* in the method. The concept of repeatability also involves making the experiment process as transparent as possible, ensuring that results can be can be checked independently by other researchers, thus raising confidence in the results (Oates, 2005; Tichy, 1998, p. 33).

Experiments fit the purpose of this thesis because we are not trying to observe or measure information technology in a social context, or in the context of human-computer interaction (HCI). Web performance metrics are in large part measurable in discrete units (load times, page weight, etc.), and therefore lend themselves to controlled experimentation. The quality of our results can be said to be measurable under four criteria: objectivity, reliability, internal validity, and external validity. This will be discussed further in the conclusion.

The experiments performed in this thesis has been structured as follows:

1. Static test web pages have been created for each performance technique tested. One page utilizing a basic implementation of an optimization technique (the *post-test*), and a second page with no technique applied (the *pre-test*). The results of a pre-test acts as a control for the results observed in the post-test.

2. All test pages have been uploaded to a web server. In the case of this thesis, shared web hosting has been provided by Domeneshop[5]. The server used was located in Oslo, Norway.

3. Both pre- and post-test pages for each technique have then been run through a test

[5] `https://www.domeneshop.no/about.cgi`

setup. The test setup was provided by WebPagetest[6]. The test setup includes a mobile device instance (both Apple's iPhone 4 and Google's Nexus 5 devices have been used) and a desktop computer instance. Both pre- and post-tests have then been run through the test setup 5 times. The median results of each have then been collected.

4. Based on the performance metrics collected from the median test runs, a comparison of the results from the pre- and post-tests have been performed.

1.3 Thesis structure

Chapter 2 explains some theoretical background on the subjects of responsive web design and web performance. Chapter 3 explains the theoretic background for the different techniques tested, highlighting their potential advantages and drawbacks. This is done to get foundational knowledge of the different techniques, so that we are more capable of predicting expected results, or explain actual results, of utilizing said techniques. In chapter 4 we discuss the results of testing the techniques covered in chapter 3, and we look at how the results correspond with the theoretic foundation. Chapter 5 will summarize the results of the experiments and compare how they correspond with the theoretical background. We will also discuss how our results contribute to answering our defined research questions. Chapter 6 contains a short summary and suggestions for further research.

[6] `http://www.webpagetest.org/`

2 Theoretical foundation

2.1 History and development of responsive web design

This section will discuss the main concepts surrounding *responsive web design* (commonly abbreviated RWD) as well as the closely related development strategies *progressive enhancement* and *mobile first*.

2.1.1 Definition of RWD

In his widely cited article, Marcotte (2010) discusses a new approach to structuring web content, an approach he labels "responsive web design". In the article, he talks about how an emerging discipline within architecture, named "responsive architecture", is trying to address some issues common with modern web design. Responsive architecture seeks to find ways for physical structures to respond to the presence of people and their surroundings. Marcotte gives a few examples: walls that bend and expand to accommodate larger crowds of people, and "smart glass" rooms that turn opaque at a certain person-density threshold (for use in business meeting rooms etc.). The principle gathered from responsive architecture suggests that "inhabitant and structure can and should mutually influence each other" (Marcotte, 2010). This principle can be transferred to web design: the constraints inherent in a device used to view web content can, and should, influence the presentation of said content.

Marcotte identifies a fluid grid, flexible images and CSS media queries as three of the central aspects of adaptive layouts. CSS3 media queries is an extension of the media type rules introduced in CSS2. The media type rules allowed different style sheets to be loaded based on which conditions a web page was viewed, for instance using different style sheets when printing a page than when viewed on a screen (Bos et al., 2009). Media queries extend this functionality by targeting more specific features than just the media type. Important in the context of responsive web design, is the fact that media queries allow

developers to target specific browser viewport[7] widths with CSS. By using properties such as `max-width` and `min-width` a media query can specify multiple rules that adapt a web page's presentation based on viewport widths (Marcotte, 2011). Media queries can likewise also target absolute screen sizes, screen resolutions, and device orientations.

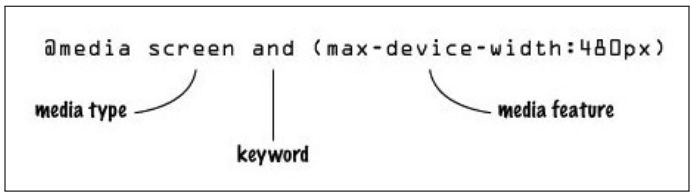


Figure 2.1: Media query syntax (UVD Blog, 2010)

The goal of RWD is to create web pages that are "context-aware". By being aware of device constraints, web pages can adjust the presentation of content accordingly. More succinctly put, responsive web design is an "approach that suggests that design and development should respond to the user's behavior and environment based on screen size, platform and orientation" (Knight, 2011). The main benefit of using RWD is that website code bases can be consolidated. Since content presentation is scaled dynamically to conform to device constraints, the need for separate websites catering to different devices is removed.

2.1.2 Progressive enhancement

Progressive enhancement is a term coined by Champeon and Finck (2003) to embody a development strategy that design web pages for the "least capable or differently capable devices first, then moves on to enhance those documents with separate logic for presentation, in ways that don't place an undue burden on baseline devices but which allow a richer experience for those users with modern graphical browser software". This approach is different from that of graceful degradation, which is a term similar to the notion of fault tolerance (Rouse, 2007). Graceful degradation focuses on building websites

[7]The visible part of a web page within a browser window.

for the most advanced browsers first, only ensuring a minimal user experience for older browsers. Progressive enhancement on the other hand focuses on "content first". This means prioritizing rich semantic HTML-markup, which ensures that content can be used even without applying presentational markup (CSS) (Gustafson, 2008b). A "content first" approach involves making sure that content is accessible and usable by anyone, regardless of location, device capabilities, or user handicaps (Gustafson and Zeldman, 2011). The browser displaying content is only served the code it is able to interpret. One way to achieve this is through modularization code (Gustafson, 2008a).

2.1.3 Mobile first

Like progressive enhancement, Mobile First is more development strategy than collection of coding guidelines. Mobile First seeks to ensure that the most important features of a web page are developed under the constraints of mobile devices, before targeting desktop computers.

Constraints are features such as screen size, hardware capabilities, and the environments (time and location) in which smart phones are used . When the first smart phones with standards-compliant browsers arrived on the market, their screen widths were approximately 80% smaller than an average computer monitor (Wroblewski, 2011, chap.2). This meant a lot less "screen real estate" for web pages, something which web developers had to take into consideration when designing mobile websites. In Mobile First the lack of screen space forces design to concentrate on essentials first. This can mean stripping away certain modules and navigation, in order to focus the user's attention towards the main content.

The Mobile First methodology has gained additional attention since the sale of mobile phones passed that of desktop computers in the end of 2010, two years earlier than predicted (Weintraub, 2011). As a result, large commercial corporations such as Google, Facebook and Adobe have announced increased focus on mobile development (Wroblewski, 2011). Some, like Google, have adopted Mobile First as an official development

strategy (Hardy, 2010). Reports show that mobile data traffic grew by 70% from 2011 to 2012, and predicting that number to increase 13-fold by 2017. It is also predicted that mobile data traffic from handsets (i.e., smartphones) will exceed 50% of all mobile data traffic in 2013 (Cisco, 2013). These numbers may give some explanation as to why Mobile First and RWD have become leading trends in front-end web development.

2.2 Browser networking and performance

This section will cover some fundamental concepts of client-server communication on the web, mobile network conditions, as well as browser processing. It will also briefly discuss why performance is important from a user perspective.

2.2.1 The web request - Fetching data from the server

To understand web performance it is beneficial to talk briefly about the main communication protocol on the web, namely the HyperText Transfer Protocol (HTTP). HTTP is the protocol that web servers and clients (usually web browsers) use to communicate with each other over the Internet. HTTP is made up of requests and responses; a client sends a request for a certain address (URL), when the server hosting this address receives the request it sends back a response (Souders, 2007, p. 6). There are several requests methods specified in HTTP, but for the sake of simplicity only the most common one, the GET-method, will be discussed. A GET-request is what downloads an asset from a URL. Usually when typing a URL in a browser address bar, what we are doing is sending a GET-request for an HTML document (e.g. `index.html`). Fig. 2.2 shows GET-request and server response headers for `www.example.com`.

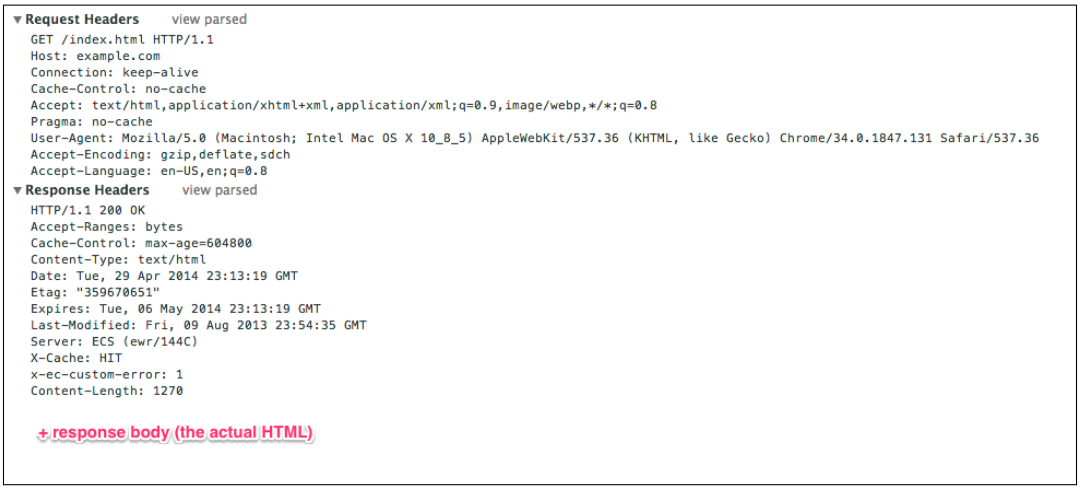


Figure 2.2: GET request and response headers for `www.example.com` (viewed in Google Chrome developer tools)

The GET-request specifies the URL of the resource it is fetching, as well as headers containing additional information, such as the protocol used, accepted file formats and encodings, and User-Agent string (which, among other things, identifies browser version). The server then responds with a status code (200 means the request succeeded), headers, and a response body. The response body in this case is the actual HTML. If the HTML contains references to external resources like style sheets, JavaScript files, or images, separate GET-requests will be issued to fetch these resources as well. All resources will be downloaded, and parsed in the case of CSS and JavaScript, before the page is considered fully loaded. When all content is downloaded and parsed, the User-Agent (browser) dispatches what is known as a `load` event. By measuring the time until the `load` event is triggered we are able to monitor how long it takes to download and process all resources on a web page.

To gain an even better understanding of performance, it is also important to look at what happens before, and during, a GET-request. Before a GET-request can be made, the client needs to resolve the domain name with a *domain name server* (DNS). Domain name resolution is the process of translating a domain name (`www.google.com`) to an

actual server IP-address (`173.194.32.52`). After domain name resolution the client and server need to open a TCP connection[8]. In order to open a TCP-connection, the client and server must first initiate what is known as a *three-way handshake*. The three-way handshake sets up connection-specific variables needed to start the data stream between client and server. First after the three-way handshake is completed can the client issue a GET-request. By this time, communication between client and server has completed a full "round trip". This can be seen by the blue lines in fig. 2.3. The time it takes for the client to contact the server, and for the server to respond, is known as the *round trip time* (RTT). The RTT is governed by several factors: the geographical distance between the client and the server, the quality of the line (e.g. copper wire or fiber-optic cable), the routing of the signal, as well as a host of other factors. In fig. 2.3 the RTT is optimistically set to 56 ms, the time it takes light to travel back and forth between New York and London.

After the initial delay caused by the three-way handshake process, the client can send a GET-request to the server. Unfortunately the server cannot send the entire response in one transmission because of a mechanism known as *TCP slow start*, which was created to prevent congestion of the underlying network. TCP slow start forces the server's first response to be small. The response size is determined by an attribute known as the *congestion window size*(cnwd). The initial cnwd was originally limited to 1 network segment, but has since been increased to 4, and in 2013 to 10. Only when the client has received, and acknowledged receiving, the initial response can the server increment the cnwd size to send more segments of data at once. On subsequent transmissions the cnwd size is doubled until maximum throughput for the underlying network is reached.

The consequence of TCP slow-start is that even if the file we are trying to download from the server is relatively small, it could still require several round trips between client and server in order to finish the download. In fig. 2.3 we see that it takes 264 ms, and a total of 4 round trips in order to download a single 21,9KB file. This process is required

[8]Transmission Control Protocol (TCP) is a transport protocol that HTTP can use to stream data between client and server.

for each new TCP connection opened between the client and server. Since opening new connections is costly with regards to performance HTTP 1.1 allows the reuse of TCP connections, meaning several resources can be sent through the same connection. This reduces the number of connections that have to be opened, and less time wasted on initiating three-way handshakes and incrementing cnwd sizes. But even with the reuse of TCP connections we still have to consider the performance cost incurred by the RTT for each network packet, which can impact download times substantially. This is one of the reasons why minimizing the total number of HTTP-requests is an important step towards better web performance (Grigorik, 2013b, Chapter 2).

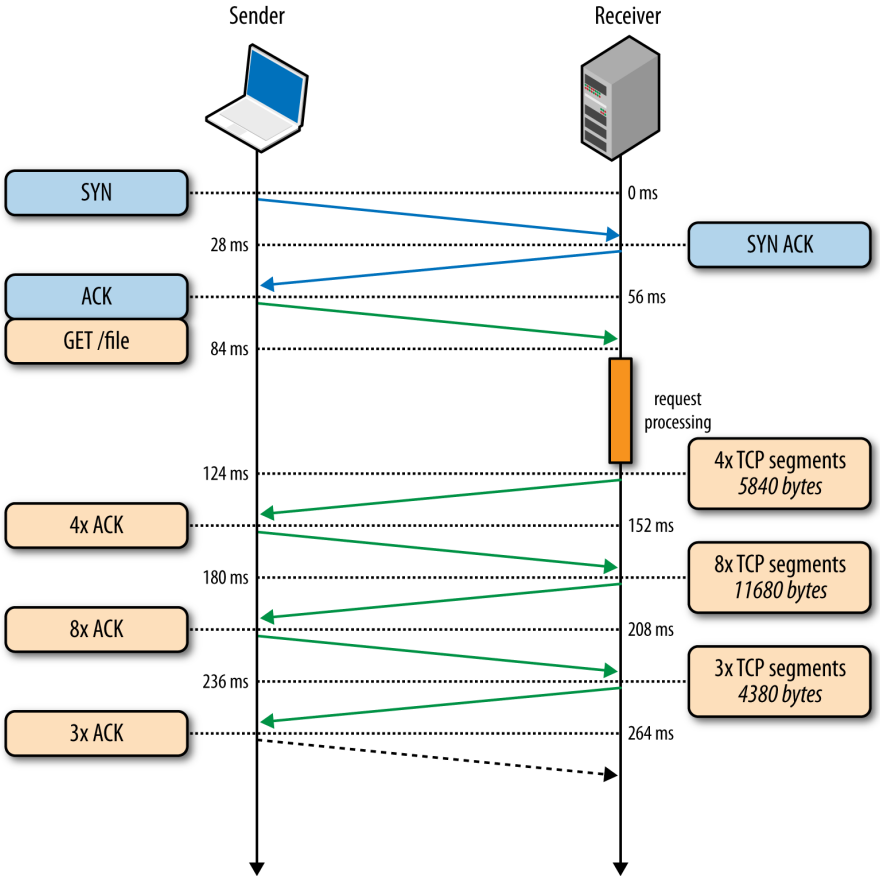


Figure 2.3: Opening a new TCP-connection, followed by GET request and response from server (Grigorik, 2013b, Chapter 2)

Other protocols than HTTP 1.1 are used for web traffic, but these are minuscule in comparison. SPDY, a protocol developed at Google, is said to improve many of the shortcomings of HTTP 1.1, but as of April 2014 only 0,7% of websites utilize it (W3 Techs, 2014). It is therefore reasonable to look at performance primarily with regards to the HTTP 1.1 protocol.

2.2.2 Mobile network latency

As mentioned, the hardware limitations of mobile devices can negatively impact performance. But the biggest challenge with regards to mobile web browsing might be the delays inherent in mobile network traffic. The content of this section is based on the research of Grigorik (2013b, chap. 7) unless otherwise stated. Fig. 2.4 shows a simplified graphic illustrating the situation.

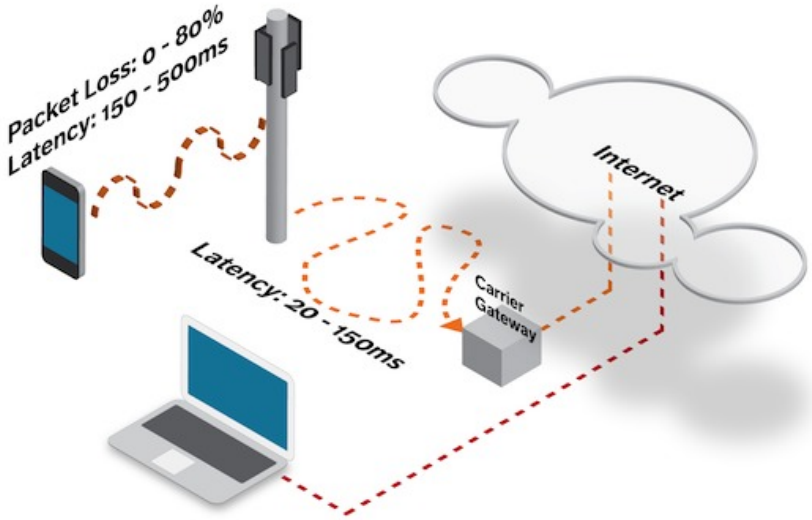


Figure 2.4: 3G and 4G network carrier latency and packet loss (McLachlan, 2013)

As illustrated by fig. 2.4, web communication over a mobile network is subject to more intermediary steps than a cabled or WiFi connection. Before a mobile device can issue a GET-request it needs to establish a connection to a nearby radio tower. The latency

imposed by establishing a connection between device and radio tower for different technologies is illustrated in table 2.1. Since the signal strength of the connection between a device and a cell tower can be highly variable, there is also a higher risk of packet loss. Packet losses incur retransmissions, which can further increase latency. After the radio tower receives an HTTP-request it needs to reroute it to a carrier gateway, which then propagates the request to the server. Because of this process, the total latency overhead for a single HTTP-request can be as much as 200–3500 ms for 3G technologies, and 100–600 ms for 4G technologies.

Generation	**Data rate**	**Latency**	**Description**
1G	no data	–	Analog systems
2G	100–400 Kbit/s	300–1000 ms	First digital systems as overlays or parallel to analog systems
3G	0,5–5 Mbit/s	100–500 ms	Dedicated digital networks deployed in parallel to analog systems
4G	1–50 Mbit/s	<100 ms	Digital and packet-only networks

Table 2.1: Data rates and latency for an active mobile connection (Grigorik, 2013b)

In addition to the latency of mobile networks, mobile devices also have to account for constrained battery life. To conserve energy, mobile devices will therefore power down their radio antennas when network connection is inactive over longer periods of time. Re-establishing a radio context can therefore introduce additional performance overhead if the device needs to power up the antenna from an idle state.

One key to faster communication over mobile networks is the utilization of of high-speed 4G technology. As shown in table 2.1, latency can be reduced to below 100 ms through a 4G connection. However, the adoption of 4G technologies is hindered by cost. This is because 4G requires a new radio interface and separate infrastructure from established 3G technologies. Grigorik (2013b, chap. 7) notes that "it takes roughly a decade from the first specification of a new wireless standard to its mainstream availability in real-

world wireless networks". Based on this estimate, widespread adoption of 4G technologies might therefore not be a realization until 2020.

It is also important to note that many smart phones and tablets were manufactured without the ability to communicate through 4G technology. Network communication over 4G is therefore dependent both on network availability/architecture and device capabilities.

2.2.3 Browser processing

This section gives a simplified overview on how the browser operates when rendering a web page. It also discusses the concept of the *critical rendering path*.

Once the HTML file for a web page is downloaded it is processed by the browser's *HTML parser*. It is the HTML parser's job to interpret the content of the HTML and translate it into a tree-structure. This structure is what is known as the *Document Object Model*, most commonly referred to as the DOM. The DOM is a representation of the hierarchy of elements within a web page, it's logical structure. Fig. 2.5 shows how the parser interprets the HTML to generate a DOM tree.

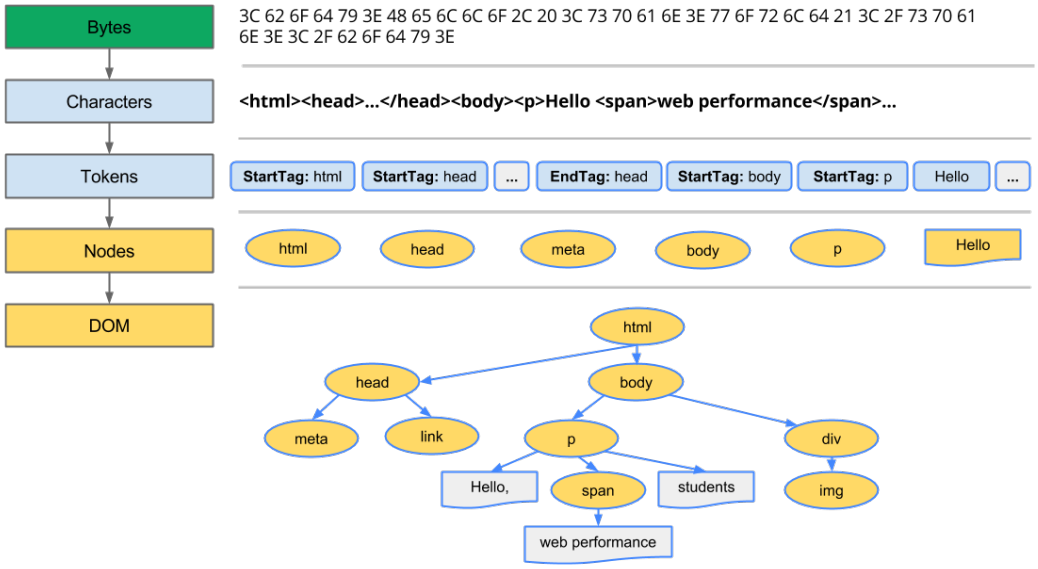


Figure 2.5: The DOM construction process (Google, 2014b)

As the parser constructs the DOM, it might encounter references to external resources such as style sheets, scripts and images. When encountering external style sheets and images, the parser will tell the browser to start fetching these resources before resuming DOM construction. It is important to note that before the browser can actually start to render anything to the screen, it needs to download and parse all style sheets within the HTML document. The parsing of style sheets the CSS Object Model (CSSOM), which contain all the collected style rules. As soon as the CSSOM is constructed, the browser can start structuring the layout paint the elements available in the DOM to the screen. The DOM does not have to be fully constructed before rendering starts, but the CSSOM does (Garsiel and Irish, 2011). Rendering is blocked until CCSOM is finished to prevent a flash of unstyled content (FOUC).

If the HTML parser encounters a script, DOM construction is halted completely. This is because scripts have the ability to inject content into the DOM, for instance through the `document.write()` method (fig. 2.6). Scripts are in turn dependent on the CSSOM because they might query for computed styles on objects in the DOM. Once the CSSOM

is ready, elements in the DOM can start to render, but if DOM construction is paused waiting on script execution, only DOM elements created up to that point can be rendered. Scripts can therefore hinder both DOM construction and rendering of content. Because of the relationship between HTML, style sheets and scripts, it has become common practice to declare style sheets in the beginning of HTML documents, and scripts at the end. By doing so ensuring that the style sheets are parsed early, enabling rendering to start as soon as possible. Putting a script at the bottom of HTML prevents it from blocking most of the DOM construction, which aides faster rendering.

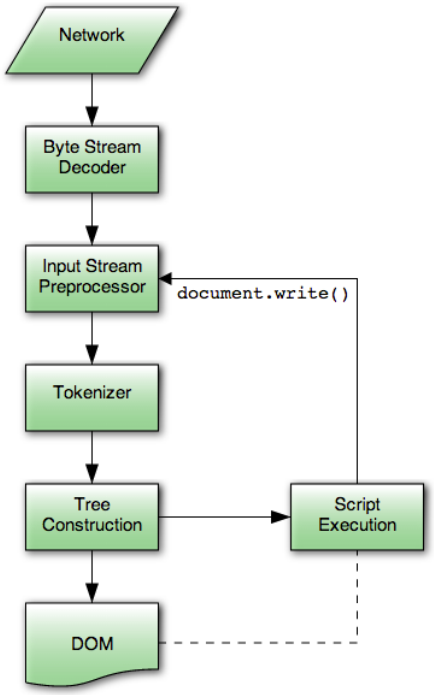


Figure 2.6: HTML parsing flow according to HTML5 spec (Berjon et al., 2014)

The critical rendering path is term defined as "the code and resources required to render the initial view of a web page" (Google, 2014a).Fig. 2.7 shows a generalized browser rendering path.

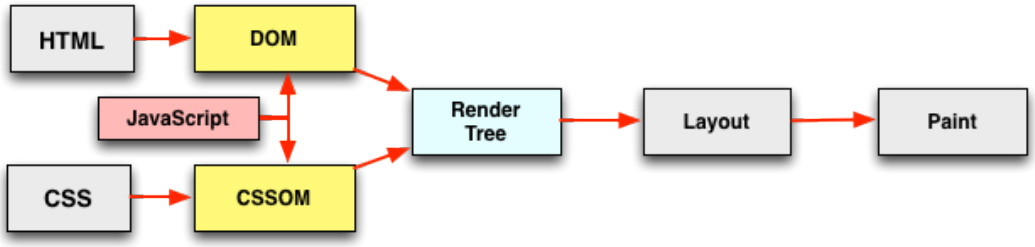


Figure 2.7: The rendering path for a website (Grigorik, 2013b)

Older browsers will halt downloading of all other resources when scripts were encountered in the DOM. Newer browsers are able to continue downloading other resources in the background using an additional lightweight parser. This parser is called the *speculative parser* (also known as the preload scanner or pre-parser). The speculative parser's job is not DOM construction or parsing of style sheets and scripts, but rather to initiate resource downloads while the main parser is waiting on a script. Speculative parsing allows the browser to continue downloading resources in parallel, even as DOM construction is halted on script parsing and execution (Davies, 2013b; Law, 2011).

Newer browsers can also use the HTML5 `defer` and `async` attributes on `<script>` elements to explicitly state how they should be handled. The `defer` attribute was the first of the two to be implemented in browsers. When encountering a `defer` script, the browser starts downloading the script, but will still continue DOM construction. Just before the DOM is finished (and the `DOMContentLoaded` event is triggered) the script will be executed. Scripts using the `async` attribute are similar, but instead of waiting until the DOM finished, they execute as soon as they are downloaded (but without blocking DOM construction) (Archibald, 2013). The `async` attribute allows scripts to be downloaded and instantiated early, without blocking DOM construction or screen rendering. `async` and `defer` are especially useful for scripts that are not used to generate the user interface, such as those used for analytic purposes.

2.2.4 The impact of performance on users

Table 2.2 below shows how users perceive delays when interacting with web pages. These numbers correlate with findings by Nielsen (1993) that state three important time limits when optimizing web and application performance. These three limits are: 0,1 seconds, instant reaction; 1 second, the limit for the users flow of thought to stay uninterrupted; and 10 seconds, the limit for keeping the user's attention focused on the process. According to Nielsen, response times should be kept under 1 second in order to keep the user's focus on the task at hand. Grigorik (2013b, Chapter 10) is even more aggressive, stating that visual feedback should be given in under 250 milliseconds in order to keep the user engaged.

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

Table 2.2: Loading time and user's perceived performance (Grigorik, 2013b)

2.3 Performance challenges handling images in responsive design

This section will discuss performance challenges connected to the use of images in responsive designs.

2.3.1 Images in responsive designs

> We have foreground and background images. We have large and small displays. We have regular and high-resolution displays. We have high-bandwidth

> and low-bandwidth connections. We have portrait and landscape orientations.
>
> Some people waste bandwidth (and memory) by sending high-resolution images to all devices. Others send regular-resolution images to all devices, with the images looking less crisp on high-resolution displays.
>
> What we really want to do is find the holy grail: the one solution that sends the image with the most appropriate size and resolution based on the browser and device making the request that can also be made accessible. (Weyl, 2013)

The quest for an ideal responsive image solution has been ongoing for several years, in fact it has been one of the biggest challenges for the RWD paradigm since the term came into widespread use around 2010 (Marquis, 2012). This section explains the reason why the use of images is especially complicated in responsive designs.

Statistics from February 2014 indicate that the weight of an average web page is 1687KB when loaded on a desktop computer. 1040kB of this total weight consists of image resources (The HTTP Archive, 2014). This means that roughly 62% of static[9] resources loaded on web pages today are images. In comparison, the ratio of image resources in November 2010 was at approximately 59%, 416kB of 702kB total (The HTTP Archive, 2010). This indicates that not only are images responsible for the bulk of page weight, but that this ratio also seems to be increasing.

Test performed by Podjarny (2013) confirm these notions, stating that images are consistently the biggest component on a page, regardless of screen size. Fig. 2.8 shows the different components of page weight for a collection of 471 responsive websites.

[9]The word static is used here to differentiate from dynamic content such as streaming audio and video, which is not counted in this statistic

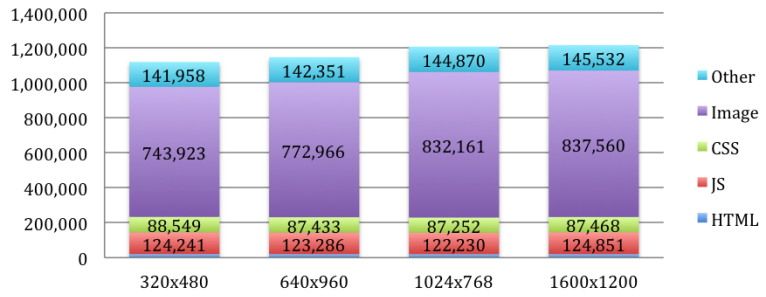


Figure 2.8: Responsive website page sizes by file type (Podjarny, 2013)

The figure shows that responsive sites only reduce page weight slightly on lower resolution screens. The results are consistent with previous measurements for responsive websites (Podjarny, 2012).In the test the, the number of requests per file type was also measured. Calculations showed that the total number of requests did not vary much across resolutions. Averages between the different resolutions also showed that of 41 total requests, 23 were for image resources (Podjarny, 2013). The fact that file sizes and requests are so similar across multiple resolutions can indicate that resources are not optimized for different screens and devices.

Since images account for the majority of page weight and requests, optimization techniques for images has a big basis for improvement, both with regards to load times and load sizes. The problem in the case of RWD, is that image optimization is difficult because of deficiencies in HTML markup and browser functionality. The deficiencies and possible solutions will be discussed in the following sections.

2.3.2 Use cases and requirements for responsive images

In 2013 the W3C published a note[10] listing several use-cases and requirements for standardizing responsive images. This document was collaboratively created by several working groups within the W3C organization[11]. The document, labeled as a work in progress,

[10] An informal document to promote discussion around a topic (W3C, 1996)
[11] `http://www.w3.org/`

seeks to define the characteristics of a responsive image, and in which situations responsive images are to be used(W3C, 2013b).

According to the document "a responsive image is an image that adapts in response to different environmental conditions: adaptations can include, but are not limited to, changing the dimensions, crop, or even the source of an image" (W3C, 2013b). "Environmental conditions" here imply device properties such as pixel-density; orientation; maximum screen width; or medium, such as `screen` or `print`. The document further specifies the following use cases for responsive images:

1. **Resolution-based selection**

   Offering the same image scaled at to different dimensions, then selecting which image to download based on a device's absolute screen resolution (width). Ensuring that larger displays gets sufficiently large images to maintain fidelity, and that devices with smaller displays are served smaller images which do not waste bandwidth.

2. **Viewport-based selection**

   In certain situations loading the correct image based on absolute screen resolution is not appropriate. Some contexts require images sizes to adapt to changes in viewport size. Change in viewport size happens when the browser window is resized, either manually (desktop) or by a change in screen orientation (mobile/tablet).

3. **Device-pixel-ratio-based selection**

   This use case focuses on the distinction between virtual pixels and device pixels. Virtual pixels are also known as `device independent pixels` (dips). Dips are abstract values used set sizes in code, such as HTMl, CSS and JavaScript. Device pixels on the other hand, is used to denote the distinct picture points on a device screen. The relation between dips and device pixels is not always 1:1. Never

devices have screens which maps multiple device pixels per dip, this creates what is known as a device pixel ratio (DPR). For instance, an iPhone 5 has a screen width measured at 320 dips, yet the actual device pixel value us 640. This means that the device-pixel ratio is 2. A 320 pixel wide raster image would cover the width on an iPhone 5 screen, but only use half of the available device pixels. This results in image artifacts, since the image is actually stretched across 640 pixels. Because many modern devices have have screens with a DPR above 1, a responsive image solution must also be able to select images based on a device's DPR.

4. **Art direction**

Instead of serving resized versions of the same image to different screen sizes (as in use cases 1-3), *art direction* suggests more comprehensive image manipulation.

When scaling down a large image for a mobile device, certain details inevitably gets lost. The main subject of the image might become to small, and details in the background obfuscated. Instead of simply scaling down the image, cropping could be used communicate the contents more effectively. This is exemplified in Fig.2.9.

Figure 2.9: Illustration of a use case for art direction (W3C, 2013b)

5. **Design breakpoints**

   For responsive images to be effective they need be able to operate under the same style rules which define layout. Breakpoints are media queries that specify the conditions (screen-width, viewport-width, screen resolution, etc.) when alternate style rules should be used. Breakpoints are often used to change the size and flow of elements in HTML for different contexts.

   If responsive image logic is to be stored in HTML (which is the case with the proposed `picture` element and `srcset` attribute which will be discussed in 2.3.4), it needs to be able to use the same units of measurements as the breakpoints defined in CSS. This means that both absolute units of measurement, like pixels; as well as relative units, like percentages or `em`/`rems`, should be possible to use to define responsive image breakpoints.

6. **Matching media features and media types**

   Instead of using display or viewport propertied, other features can be used to select appropriate image resources. For instance loading different images for grayscale displays (e.g. e-ink readers) than color displays, or serving higher resolution images to printers.

7. **Image formats**

   A responsive image solution should be able to specify fallback options for unsupported image formats. Some image formats might offer better compression (such as WebP) or scalability (such as SVG) than other formats. However, these formats might have limited browser support, are therefore abandoned. An image solution where fallback image formats could served to legacy browsers would alleviate this problem.

8. **User control over sources**

   A responsive image solutions should aid in giving users control over image downloading, for instance when roaming on a mobile device. The user should have the ability to change settings that make the browser downsample image quality, or refrain from loading images altogether.

From these use cases, the W3C document expresses the following requirements:

1. To allow for art direction, the solution must afford developers the ability to match image sources with particular media features and/or media types - and the user agent should update the source of an image as the media features and media types of the browser environment change dynamically.

2. The solution must support selection based on viewport dimensions, screen resolution, and device-pixel-ratio (DPR). Sending the right image for the given dimension and DPR avoids delaying the page load, wasting bandwidth, and potentially reduces the impact on battery life (as the device's antenna can be powered off more quickly and smaller images are faster to process and display). It can also potentially save users money by not downloading redundant image data.

3. The solution must degrade gracefully on legacy user agents by, for example, relying on HTML's built-in fallback mechanisms and legacy elements.

4. The solution must afford developers with the ability to include content that is accessible to assistive technologies.

5. The solution must not require server-side processing to work. However, if required, server-side adaptation can still occur through content negotiation or similar techniques (i.e.,, they are not mutually exclusive).

6. The solution must afford developers the ability to define the breakpoints for images as either minimum values (mobile first) or maximum values (desktop first) to match the media queries used in their design.

7. The solution should allow developers to specify images in different formats (or specify the format of a set of image sources).

8. To provide compatibility with legacy user agents, it should be possible for developers to polyfill the solution.

9. The solution should afford user agents with the ability to provide a user-settable preference for controlling which source of an image they prefer. For example, preference options could include: "always lowest resolution", "always high resolution", "download high resolution as bandwidth permits", and so on. To be clear, user agents are not required to provide such a user-settable preference, but the solution needs to be designed in such a way that it could be done.

10. In order to avoid introducing delays to the page load, the solution must integrate well with existing performance optimization provided by browsers (e.g., the solution would work well with a browser's preload scanner). Specifically, the solution needs to result in faster page loads than the current techniques are providing. (W3C, 2013b)

2.3.3 Responsive images using CSS

When RWD was first defined, its cornerstones were "fluid grid" layouts and "flexible images" (Marcotte, 2010). Flexible images were a simple solution to handling images in responsive designs, only requiring one line of CSS:

```
img { max-width:  100%; }
```

With this rule applied, `<img>` elements will never stretch beyond the full width of their

parent element. If the image is wider than the parent element, it will be scaled down to fit within the width of its parent, preventing overflow[12] (Marcotte, 2011, p.45). Since the `max-width` attribute was introduced with the CSS 2.1 specification, it has wide browser support, and can therefore be used with little or no inconsistencies between browsers. Using this rule will in most cases ensure that the dimensions of an image are maintained regardless the dimensions of other elements in the layout. However, the main problem with "flexible images", and one of the reasons why a responsive image solution is needed, is that it enforces a habit of using the same large images for all layouts. This poses problems for mobile devices which have to download excessively large images, often over slow and/or metered mobile networks. In addition, downscaling images can be a compute-intensive process adding additional delays to rendering. This is often especially problematic for mobile devices because of their limited graphic processing capabilities.

The background-image solution

The challenge for responsive images is how to load the right resource under differing conditions. Conditions are device and context dependent and might encompass everything from device properties (e.g. screen resolution) and viewport size, to network limitations. The conditions should be able inform the browser which images to download.

In fact, a simple solution to conditioned image selection has existed since the introduction of media queries in CSS3. However, it is limited to background images set in CSS:

[12]Content exceeding the dimensions of its parent element.

Listing 2.1: CSS for a responsive background image

```
<!-- CSS -->
<style>
/* Small image used by default if none of the media queries are triggered */
.image {
      background-image: url('small.jpg');
      width: 300px;
      height: 150px;
}

/* Medium image loaded for viewport-width > 600px */
@media all and (min-width: 600px) {
   .image {
      background-image: url('medium.jpg');
      width: 600px;
      height: 300px;
   }
}

/*Large image loaded for viewport-width > 1000px */
@media all and (min-width: 1000px) {
   .image {
      background-image: url('large.jpg');
      width: 1000px;
      height: 500px;
   }
}
</style>

<!-- HTML -->
<div class="image"></div>
```

The advantage of this solution is that only the image that matching a media query rule (or lack thereof) is loaded. After the initial load, the other images can be loaded dynamically if the viewport is resized to trigger one of the other breakpoints.

So, if this solution is able to download different images based layout breakpoints, why not use it for all images? Several reasons:

- **Prevents separation of concerns**

  Web pages should enforce a separation between presentation (CSS) and content (HTML). The majority of images on the web is considered part of content. By declaring content images in CSS the separation of concerns is broken. It is no longer possible to identify content from HTML alone.

- **Unsemantic markup**

  When not using the `<img>` element it is hard to identify an element as a content image. This also complicates the indexing by web crawlers and screen readers, which is bad for SEO[13], and usability.

- **Redundant markup and added complexity**

  Another issue with delegating images to CSS is all the additional markup it would require. Each new image would need a separate class (or ID) selector to declare the image URL, and a new URL would need to be assigned to this class for each supported breakpoint. Also, unlike inline images, background images declared in CSS have no implicit height or width. This would then have to be set explicitly like in listing 2.1. This becomes tedious and complex if a web page with many content images, and additionally so if new images are continually added. Setting image dimensions in absolute units (pixels) is also less suitable for responsive designs since it prevents image scaling. Adapting the technique in listing 2.1 to use relative units would further complexity.

Using the CSS `background-image` property for content images is therefore a design anti-pattern which should be avoided. However, as a means to handle actual background images in a responsive fashion, the technique outlined in listing 2.1 is perfectly acceptable.

2.3.4 Potential standards for responsive images

Multiple sources in `<video>` element

The `<video>` element in HTML5 introduced a new way of conditionally loading resources. The `<video>` element is used to embed video content, and uses a child `<source>` element to specify the location of this content. What is unique to `<video>` compared to `<img>`, is the fact that each `<video>` element can contain multiple `<source>`

[13]Search engine optimization

elements that can specify fallback options in case a browser does not support the default video format. More relevant with regards to RWD is the fact that `<source>` elements can be loaded conditionally by specifying a media query inside a `media` attribute (Devlin, 2012). Consider the following example:

Listing 2.2: `video` element using multiple sources and media queries

```
<video>
   <source src="video-clip.mp4" type="video/mp4">
   <source src="video-clip.webm" type="video/webm">
   <source src="video-clip-small.mp4" type="video/mp4" media="all and (max-width:480px)">
   <source src="video-clip-small.webm" type="video/webm" media="all and (max-width:480px)">
</video>
```

In this example a single video exists in four different versions: two different video formats, and two different file sizes. The default video is in the `mp4` format. The file in `webm` format is loaded as a fallback if the browser does not support `mp4`[14].

However, if the browser viewport has a width smaller than 481px, one of the smaller video files (`video-clip-small.mp4` or `video-clip-small.webm`) is downloaded instead. This shows a way in which a single element can offer multiple versions of the same content under varying conditions, offering fallback options for unsupported file formats.

Multiple sources in `<img>` element

Unfortunately there exists no native method to conditionally load images like it does for `<video>` in any existing browser, but browser vendors are actively discussing and developing many possible alternatives. Some of these alternatives include the `<picture>` element (W3C, 2013a); the `srcset` and (now abandoned) `src-N` attributes for the `<img>` element (W3C, 2013c); *Client Hints*, offering image content negotiation through HTTP headers; and various responsive image formats (Caceres, 2013). Each of these alternatives have their own idiosyncrasies, benefits, and disadvantages. At the time of writing the `<picture>` element (W3C, 2013a) and `srcset` attribute seem to have most

[14]Some browsers, like Opera and Firefox, do not support the mp4 video file format (Coding Capital).

traction. However, it is hard to say if, and when, any of these alternatives will see full implementation in browsers, and it will likely take some time before any method gains broad adaptation. Listing 2.3 shows a proposed syntax for the <picture> element.

Listing 2.3: The proposed syntax for the <picture> element (W3C, 2013a)

```
<picture width="500" height="500">
      <source media="(min-width: 45em)" src="large.jpg">
      <source media="(min-width: 18em)" src="med.jpg">
      <source src="small.jpg">
      <img src="small.jpg" alt="">
      <p>Accessible text</p>
</picture>
```

This syntax is quite similar to that of the <video> element in listing 2.2. The <picture> element, like <video>, also acts as a container for multiple <source> elements. Each <source> element references images for differing media conditions. An <img> element is also included to support legacy browsers.

2.3.5 Responsive images using existing technologies

Solutions to responsive image use cases and requirements already exist using current technologies. The problem is that these techniques "rely on either JavaScript or a server-side solution (or both), which adds complexity and redundant HTTP requests to the development process. Furthermore, the script-based solutions are unavailable to users who have turned off JavaScript.The RICG believes standardization of a browser-based solution can overcome these limitations"(W3C, 2013b). However, since browser-based (HTML-native) solutions have yet to be implemented in any existing web browser, this thesis will be evaluating the effectiveness of some of these script and server-based solutions. In doing so, advantages and drawbacks to each solution will be discussed. The aim is to explore if the use of these responsive image techniques can be warranted from a performance perspective, despite adding complexity and redundant HTTP requests to the code base.

2.4 Other performance considerations

As mentioned in section 2.3.1, images makes up the bulk of many responsive websites weight and requests. The use of images in responsive designs is therefore a prime candidate for improvement. However, this does not mean that only image-related optimization techniques should be taken into account when discussing performance. Front-end performance can be effected by all techniques that reduces requests issued by the client or the bytes downloaded from the server, as well as techniques that accelerate screen rendering. This thesis therefore also discusses other, more established, techniques for improving performance. The aim is to see if these techniques affect performance differently under mobile and desktop browsing conditions.

The techniques covered will be split into four categories: implementing responsive images, optimizing UI graphics, optimizing textual resources, and improving server conditions.

3 Review of performance optimization techniques

3.1 Implementing responsive images

As of today, some of the challenges outlined in section 2.3.2 can only be handled by CSS, JavaScript or server-side scripting, but not in a browser-native way using HTML. These techniques are workarounds that address problems that the browsers can not handle natively at the present time. Such a technique is usually called a "polyfill" or "polyfiller" (Sharp, 2010). This section discusses four existing solutions for responsive images.

3.1.1 Picturefill

Created by Scott Jehl in 2012, Picturefill is a client-side JavaScript polyfill that enable responsive images. The script is designed to allow HTML markup similar to the proposed `<picture>` element, using `<span>` elements that dynamically load images based on media queries. The following example shows some necessary markup to make the script to work:

Listing 3.1: Basic HTML markup for Picturefill (Jehl et al., 2014)

```
1  <span data-picture
2  data-alt="A giant stone face at The Bayon temple in Angkor Thom, Cambodia">
3     <span data-src="small.jpg"></span>
4     <span data-src="medium.jpg"     data-media="(min-width: 400px)"></span>
5     <span data-src="large.jpg"      data-media="(min-width: 800px)"></span>
6     <span data-src="extralarge.jpg" data-media="(min-width: 1000px)"></span>
7
8     <!-- Fallback content for non-JS browsers.
9     Same img src as the defult Picturefill img -->
10    <noscript>
11       <img src="small.jpg"
12       alt="A giant stone face at The Bayon temple in Angkor Thom, Cambodia">
13    </noscript>
14 </span>
```

All responsive image syntax is contained within a `<span>` element with a `data-picture` attribute. The `data` attribute is new to HTML5, and allows storage of user-defined data within elements. The `data` attribute can be used in cases where no other suitable at-

tribute exist (Lumsden, 2012). In the case of Picturefill, the `data` attributes are used as hooks for the `picturefill.js` script. The `data-alt` attribute stores the alternate text to be used with the image, the `data-src` attribute stores the path to the image resource, and the `data-media` attribute stores the media condition (media query) that triggers the download of an image. In this example `small.jpg` is loaded as a default if none of the specified media conditions are evaluated as true, or if JavaScript is disabled in the browser, in which case the markup inside the `<noscript>` element is used. If a media condition stored in a `data-media` attribute evaluates to true, only the corresponding image is loaded. This will ensure that only a single image is loaded at once, so that no bandwidth is wasted loading multiple versions of the same image at the same time. If the media condition were to change after the initial page load, for instance by browser resizing or change in device orientation, download of other images can be triggered. The initially loaded image will then be replaced by the image matching the current media condition. When the script figures out which `<span>` best matches the current media condition, the `<span>` is replaced by an `<img>` element, which is then populated with the correct `src` and `alt` attributes.

The reason why `<img>` elements are not used in listing 3.1 is because an `<img>` element without a `src` attribute is considered invalid HTML. Also, `<img>` elements that have a specified `src` will always trigger a download, regardless of media conditions, unless defined within a `<noscript>` element, in which case it will only download if scripting is disabled.

Advantages

- **Support many use cases**

  The Picturefill solution can be very versatile in regards to image selection. Each main `<span>` element can contain as many child `<span>` elements (image containers) as is needed to support different conditions. In addition, each `data-media` attribute can chain multiple media queries to increase specificity. Consider the

following example:

Listing 3.2: Chained media query specificity

```
1    <span data-src="specific.jpg" data-media="(min-width: 568px) and (min-
         device-pixel-ratio: 2.0) and (orientation: landscape)"></span>}
```

This condition would only download an image on a mobile or tablet device held in landscape orientation mode with a minimum viewport-width of 568 (CSS) pixels and a device-pixel ratio (DPR) of 2 or higher. Picturefill's media query support means that it supports resolution-based, viewport-based, DPR-based image selection. The fact that different images can be specified for for each media condition means that art direction is also supported.

Drawbacks

- **Verbose markup in HTML**

  By having to explicitly set image paths for and media conditions for each image, the markup required in HTML becomes quite verbose.

- **Manual image creation and referencing**

  For each image size you wish to support, an image needs to be duplicated, scaled or cropped, its path needs to be referenced, and a corresponding media query needs to be written. Doing this for each image on a page quickly becomes time consuming. However, automation of the process could probably be implemented, for instance through a content management system (CMS).

- **Breaks DRY principle**

  The DRY principle states "that every piece of system knowledge should have one authoritative, unambiguous representation" (Venners, 2003). Picturefill cannot use the same breakpoints as declared in CSS without breaking the DRY principle. The breakpoints are then declared in both HTML and CSS, changing breakpoints in one of them does not influence the other.

- **Image downloads must wait on JavaScript execution**

  The responsive image logic is dependent on the `picturefill.js` script. This means that no images can start downloading until the script is downloaded, parsed and executed. Images can therefore not be detected by speculative parsing.

- **Script-dependent**

  Users with JavaScript disabled will not be able to use Picturefill functionality. It is therefore necessary to set fallback image sources using the HTML `<noscript>` element. Making images script dependent can also affect indexing by web crawlers, possibly affecting SEO.

- **Additional HTTP-request**

  An additional HTTP-request is necessary to download the `picturefill.js` script.

3.1.2 Adaptive Images

Adaptive Images (henceforth AI) is a server-side solution for responsive images written in PHP. Originally authored by Matt Willcox in 2011, Adaptive images was created to offer a server automated alternative to responsive images.

The first thing that needs to be in place in order for AI to work, is a method that can identify the resolution of the device. This can be done by writing the resolution to a session cookie using JavaScript. The following code snippet shows one way of doing this. This snippet needs to be included in all HTML documents utilizing AI, preferably as early as possible in the markup (for instance at the beginning of the `<head>` element).

Listing 3.3: Writing the AI session cookie (Wilcox, 2012)

```
1  <script>
2     document.cookie='resolution='+Math.max(screen.width,screen.height)+'; path=/';
3  </script>
```

To use AI on a site hosted by an Apache server[15], uploading or modifying the `.htaccess`

[15] AI also runs on Nginx servers, but require a slightly different implementation.

file[16] in the root directory is required. This modified version of `.htaccess` tells the server that requests for images of certain formats (files with the extensions .jpg/.jpeg, .gif, .png, etc.) should be interpreted by the `adaptive-images.php` script. This script then compares the width of the original image embedded in the HTML with the resolution stored in the session cookie, and compares this value with an array in the script which stores the different resolutions that should be supported. If the screen width stored in the cookie is smaller than the width of the original image, the script looks at the array and finds the defined dimension closest to the width stored in the cookie. It then checks to see if this image has already been generated for this size. AI stores all downsized images in a cache folder. If the image exists in the cache folder for the correct size (i.e. ai-cache/500 for images of 500 px width) that image is fetched. If the image doesn't exist in the cache at the correct resolution, a resized version is created from the original image on the fly. The image is then stored in the appropriate cache folder, and sent back to the client.

Advantages

- **Automation**

  The main advantage to this solution is that it automates all the work of scaling and referencing the correct images for different screen dimensions. This can save a lot of time in the development and maintenance phases of a website, especially when supporting many differently sized screens. The automated process makes it easier to use this solution in cases where content authors do not write the actual markup, such as in a CMS system.

- **No additional HTML markup**

  The fact that this solution requires no additional markup to work adds to HTML

[16] ".htaccess files (or "distributed configuration files") provide a way to make configuration changes on a per-directory basis. A file, containing one or more configuration directives, is placed in a particular document directory, and the directives apply to that directory, and all subdirectories thereof."(Apache, 2014).

document legibility and to the technique's ease of use. Unlike Picturefill, where all alternate image paths has to be explicitly referenced, AI automates the process internally on the server.

Drawbacks

- **Session cookie**

  The fact that Adaptive images is dependent on a session cookie to store device-widths is one of its weaknesses. This is because there is no guarantee that the cookie will be written and sent with the first image request on the page. This has to do with how different browsers load resources such as images and scripts. Older browsers block image downloading while scripts in the <head> are downloaded and parsed (Souders, 2007, p.48), but newer browsers will use speculative parsing to load images in parallel (Weiss, 2011). If an image is requested before the cookie has been written, the server has no way of knowing the device's screen dimensions, and can therefore not rescale and serve a correctly sized image. Because of this, Adaptive images has implemented a fallback solution which reads the User-Agent string[17] for content negotiation in case the cookie is not yet stored. If the UA-string indicates a mobile device, the smallest defined image is loaded. If not, it's assumed that the device is a desktop computer, and the original image is loaded (Wilcox, 2012). Using the UA-string also allows AI to work when scripts are disabled, albeit with less precision in the image selection logic.

- **Latency on first image load**

  A visible latency is introduced if there does not exist a downscaled image matching the device width currently stored in the session cookie. The AI script then has to rescale and serve a new image on the fly. The result of this process is that the image can't be downloaded until the script is finished resizing the new image, something which adds to perceived delay. This is a minor drawback since the image would

[17] A text field in an HTTP request header that contains the name, version, as well as other features of the web browser

only need to be requested once in order to be resized and cached. The delay is therefore only present the first time an image is requested for any supported screen size.

- **Lack of breakpoint specificity**

  AI only queries for absolute device screen resolutions, it is therefore unable to load different images for changes in other conditions, such as viewport size. One use case where the benefit of AI is lost, is when the layouts for larger resolution screens actually require smaller images than layouts for lower resolution screens. This can be the case when a web page utilizes a multi-column layout on larger screens, which then adapts into a single column for smaller (mobile) screens. See Fig.3.1 for a more detailed illustration.

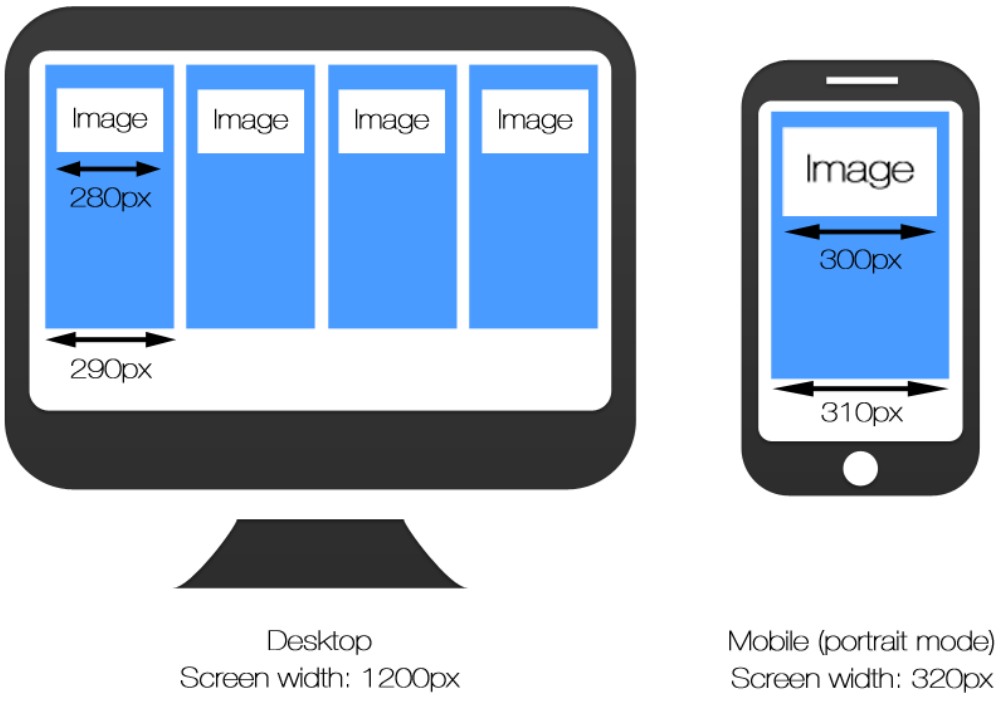


Figure 3.1: Differing column layouts for mobile and desktop environments

The illustration shows a four column layout for desktop browsers, in this case the horizontal resolution of the monitor is 1200px. Adding in some space for padding

we see that images within the columns never stretch beyond 280px in width. On a mobile device held in portrait mode, the layout has adapted to a single column. In this case the mobile device has a horizontal resolution of 320px, and the layout requires images that are 300px wide. For the sake of this example, let us say that AI was set up to serve images at 320px and 1200px. In this case the desktop layout would get served 1200px wide images, whereas the mobile layout would get served 320px wide images. This illustrates the problem where the device screen-width resolution does not inform the selection of the most appropriate image size for the given layout. In this case, the lack of customizability is a weakness.

3.1.3 Clown car technique

So far, two approaches to responsive images have been discussed: one client-side approach using JavaScript, and one server-side approach using PHP. To round out the discussion on responsive images we wish to look at another client-side approach labeled "the clown car technique". The clown car technique (henceforth shortened CCT), like AI, removes the image-selection logic from the HTML or CSS. The technique does this by leveraging "media queries, the SVG format and the <object> element to serve responsive images with a single request" (Weyl, 2013).

Unlike Picturefill, CCT is not strictly a polyfill solution as it uses existing technology to adaptively load images (although in an unconventional fashion). The foundation of CCT lies within the SVG format. By definition SVG "is a language for describing two-dimensional graphics in XML. SVG allows for three types of graphic objects: vector graphic shapes (e.g., paths consisting of straight lines and curves), images and text"(W3C, 2011). SVG is perhaps most well-known for being a vector graphic format, but as stated by the preceding quote it may also be used to define raster images and text. Styling can also be applied to SVG through either XSL or CSS. CCT leverages the abilities of SVG by constructing media queries within an SVG file, which selectively load raster images as backgrounds in CSS. Listing 3.4 gives an example of such an SVG file.

Listing 3.4: SVG file used in by CCT (Weyl, 2013)

```
<svg xmlns="http://www.w3.org/2000/svg" viewBox="0 0 300 329"
    preserveAspectRatio="xMidYMid meet">
<title>Clown Car Technique</title>
<style>
svg {
   background-size: 100% 100%;
   background-repeat: no-repeat;
}
@media screen and (max-width: 400px) {
   svg {
      background-image: url(images/small.png");
   }
}
@media screen and (min-width: 401px) and (max-width: 700px) {
   svg {
      background-image: url(images/medium.png);
   }
}
@media screen and (min-width: 701px) and (max-width: 1000px) {
   svg {
      background-image: url(images/big.png);
   }
}
@media screen and (min-width: 1001px) {
   svg {
      background-image: url(images/huge.png);
   }
}
</style>
</svg>
```

Here the SVG file is essentially functioning as a container for all the image-selection logic. The file could then be embedded in HTML by declaring:

```
<object data="responsive-image.svg" type="image/svg+xml" ></object>
```

The reason why the `<object>` element is used instead of the more semantically correct `<img>` element, is because of inconsistencies in how different browsers handle SVGs referenced in `<img>`. Some browsers do not allow external images to be loaded, while others fail to preserve the images' aspect ratio set within the SVG file, when SVG is referenced in `<img>`. None of these issues occur when using `<object>` (Weyl, 2013). Using CCT, images can be selectively loaded similarly to when using Picturefill, but with some important distinctions. The media queries specified within the SVG file are not relative to the browser, but rather to its parent element. For instance, if a CCT

`<object>` is declared within a `<div>` element, image selection would be dependent on the properties of the `<div>`. Unlike Picturefill, which loads inline images, images specified within the SVG in CCT are loaded as background images. As stated in section 2.3.3 background images need their dimensions explicitly set in order to be visible. Within SVG, aspect ratio and scaling is defined by the `viewbox` and `preserveAspectRatio` attributes. Declaring `background-size: 100% 100%;` on the SVG itself ensures that the image stretches to the full width of the object element.

CCT does not utilize HTML markup or CSS to chose images. Instead this logic resides in the style definitions of a SVG file, which pulls in images as backgrounds at different breakpoints. The separation of image selection logic from HTML and CSS can be seen as a feature, but at the same time it necessitates a separate SVG file for each image embedded in HTML. The SVG file needs to be downloaded and parsed before an image can be downloaded, adding latency and an additional HTTP request for each image.

Advantages

- **Client-side logic not requiring JavaScript.**
  Since this technique doesn't utilize JavaScript it will still work even if scripting is disabled.
- **Removes image selection logic from HTML markup.**
  Since the image selection logic resides in the SVG file, the HTML markup becomes less cluttered than when using Picturefill.
- **Images adapt to the size of parent element**
  CCT is the only responsive image solution discussed in this thesis where the breakpoints are not relative to the size of the screen or viewport. Instead it is relative to its parent element. This allows the responsive images to be more modular, relative to other elements on the page. For instance, on a 2000px wide screen, an image might reside inside a layout column which never stretches beyond 500px wide. If the responsive image breakpoints are relative to the screen or viewport in this situ-

ation, the image downloaded would be much larger than necessary. By using CCT, the image would be chosen relative to the parent element, which would be 500px, and not 2000px. The big trade-off in this situation, is that image selection logic becomes dependent on style calculations, and therefore has to wait on CSS.

- **Supports many use cases**

  As with Picturefill, this solutions supports many responsive image use cases. The exception being viewport-based selection, where CCT instead supports "parent-element-based selection".

Drawbacks

- **SVG not supported by older browsers**

  SVG is not supported in older browsers, such as IE prior to version 9.0 and Android browser prior to version 3.0 (Can I Use, 2014).

- **Code redundancy**

  Even though the technique requires little additional HTML markup, the SVG markup required is quite extensive. In order to be maintainable this step would need to be automated.

- **Breaks DRY principle**

  Like Picturefill, CCT breaks the DRY principle by declaring breakpoints in both CSS and SVG. However, since breakpoints in SVG are relative to parent elements they might differ from CSS breakpoints.

- **Unsemantic markup**

  Replacing the `<img>` element with the `<object>` element removes semantics from the HTML. The fact that images are loaded as backgrounds in CSS complicates this issue further.

- **Image size must be explicitly stated**

  Since images are loaded as backgrounds the SVG, they needs to have aspect ra-

tio declared in `viewbox` attribute in order to ensure that image dimensions are displayed correctly.

- **Additional HTTP-requests**

  Requires additional HTTP-requests to download SVG files before images can be downloaded. It is possible to inline the entire SVG files using base64 encoding, but this will again bloat the size of the HTML file considerably.

3.1.4 Lazy loading

A good way of saving bandwidth is to delay loading of non-critical resources. One way this can be achieved is through delayed resource loading, also known as lazy loading. Lazy loading is a technique used to only load resources that are displayed within the browser viewport on initial page load. Loading of other embedded resources (also known as "below the fold" content) is then delayed until a placeholder is scrolled into (or near) the viewport area. In this thesis we will be focusing on lazy loading images, but the concept encompasses any asynchronous fetching of content or resources (such as delayed content retrieval through AJAX). Lazy loading, on its own, is not a responsive image solution, but the two can easily be combined

Lazy loading images is usually done through client-side scripting. A common approach is to store the actual image path in a `data` attribute within the `<img>` element:

```
<img class="lazy" data-original="img/example.jpg" alt="Lazy loaded image"
```

Using JavaScript event handlers, it is then possible to measure when a user scrolls the `<img>` element with the class "lazy" into (or close to) the viewport. The script can then insert the path stored in `data-original` into a `src` attribute. This will trigger an image download. Omitting the `src` attribute in `<img>` is considered invalid HTML markup. This issue can be circumvented by using a small placeholder graphic which is then replaced by the proper image when lazy loading is triggered.

Lazy loading can reduce the number of requests needed when initially loading a web page. Less resource requests improves load times and can improve page rendering speed, thereby increasing perceived performance. Unlike hiding elements with CSS (using `display: none;`), lazy loading prevents resources from being loaded entirely if they appear outside of the browser viewport (below the fold). This can be especially effective for images considering their relatively large file sizes compared to textual resources. As stated in section 2.3.1 over 60% of average web page weight come from images. Delaying the loading of images below the fold can therefore greatly reduce initial page weight, as well as reduce load time.

Advantages

- **Reduces HTTP-requests on initial page load**

  Requests will only be made for images within the viewport on initial load. Other requests are deferred until placeholder is scrolled within viewport. Less HTTP-request lead to faster initial load times.

- **Reduces page weght on initial page load**

  Deffering image downloads can reduce much of the page weight on initial load. This because images usually comprise the bulk of average web page weights. Smaller page weight can in turn lead to faster load times, especially on low bandwidth connections such as mobile networks.

- **Ensures that bandwidth isn't wasted loading images that are not viewed**

  If a user navigates away from a page without viewing it in its entirety, lazy loading ensures that bandwidth isn't wasted loading images that are never viewed.

Drawbacks

- **Lazy loaded images trigger content reflow**

  By default the image placeholder will not reserve any space for the image that is to be lazy loaded. This will cause other content to shift around when an im-

age is lazy loaded into the layout. This happens because the browser needs to reflow elements on the page in order to accommodate space for the new image. In a responsive design the image inserted would be dynamically resized using the `max-width:100%` rule in CSS. Therefore space cannot be reserved by hard-coding `width` and `height` attributes in the `<img>` element.

The combination of recalculation of DOM elements and image sizes together when lazy loading can add considerable delays in rendering. The reflow of elements in the layout may also result in a degraded user experience. A case study performed Andersen and Järlund (2013) concluded that image reflows were a big challenge with regards to user experience and usability. Wilton-Jones (2006) states that "[r]eflows are very expensive in terms of performance, and is one of the main causes of slow DOM scripts, especially on devices with low processing power, such as phones. In many cases, they are equivalent to laying out the entire page again". The case study by Andersen and Järlund (2013) gives one possible solution to this problem. The performance and usability cost of reflows can be remedied by adding CSS, levering intrinsic ratios in what is also known as the "padding-bottom hack". Listing 3.5 shows how it works.

Listing 3.5: Using the "padding-bottom hack" to constrain image proportions (Andersen and Järlund, 2013)

```
.img-container {
   position: relative;
   padding-bottom: 56.25%; /* 16:9 ratio */
   height: 0;
   overflow: hidden;
   background-color: black;
}

.img-container img {
   position: absolute;
   top: 0;
   left: 0;
   width: 100%;
   height: 100%;
}
```

Adding the class `img-container` to an element (e.g. `<span>` or `<div>`) that

wraps around a lazy loaded image , ensures that an area is reserved by the value set in its `padding-bottom` property. The area that is reserved is determined by the ratio enforced by the `padding-bottom` value. Setting it to 56,25% reserves an area with the common ratio 16:9. The ratio can be adjusted by changing the value of `padding-bottom` (e.g. 100% would give the ratio 1:1). The width of the `.img-container` element is dependent on the width of its parent element, while child `<img>` elements of `.img-container` will cover the reserved area (since `width` and `height` is set to 100%). The CSS in listing 3.5 will give the results shown in fig. 3.2.

Figure 3.2: The effect of the `padding-bottom` hack (Andersen and Järlund, 2013)

The padding bottom hack has at least two caveats that need to be considered:

1. The ratio (relation between height and width) for images needs to be known in advance in order to apply the correct `padding-bottom` value to the wrapper class (`.img-container`). Multiple container classes can be made to accommodate for different ratios (e.g. `class="img-container ratio-16:9"` or `class="img-container ratio-1:1"`) .
2. Since image width is stretched to the full width of the `.img-container` element, you run the risk of expanding the image beyond its native size.

If a website mostly uses images with the same aspect-ratios, the first point should

not be a big challenge. However, if image aspect-ratios cannot be controlled, creating applying correct container classes would be much harder. The second point would become an issue if the loaded image is smaller than its parent element, care need therefore be taken to ensure that sufficiently large images are present at all breakpoints in the layout.

- **Might require additional TCP-connections**

  Since some images will be loaded later than others it is possible that TCP connections will be closed before all images are downloaded (depending on how fast a user scrolls down a page). This results in new connections having to be made to download new images. Establishing new connections has an inherent delay (as discussed in section 2.2.1) and is less effective than reusing existing connections.

- **Script-dependent**

  Users with JavaScript disabled will not be able to lazy load images. It is therefore necessary to set fallback images using the HTML `<noscript>` element. Making images script dependent can also affect indexing by web crawlers, possibly affecting SEO. Image downloads are also dependent on script execution. It might therefore make sense to avoid lazy loading images early in the HTML document, since these images might appear above the fold on initial page load.

- **Additional HTTP-request**

  An additional HTTP-request is necessary to download the lazy loading script.

3.2 Optimizing UI graphics

3.2.1 Image sprites

As previously stated, the number of HTTP-requests required to load a web page's resources can have a big impact on web page performance. This issue is further amplified on mobile networks because of the inherent delays connecting to cell towers. A common optimization technique is therefore to attempt to reduce the number of requests as much

as possible. One way this is done is by concatenating several resource into single files. Image sprites are a way of combining multiple images into a single larger image. When an image sprite is loaded, individual images can be referenced in CSS using their respective coordinates within the image sprite. The following example demonstrates a simple use case.

Four 10 by 10 pixel images illustrate arrows that go in different directions. Instead of loading each image, creating four HTTP-requests, the images can be combined into a single image file and used as follows:

Listing 3.6: Simple usage of image sprite

```
<!-- CSS -->
<style>
.arrow {
   background-image: url(arrows.png);
   height: 10px;
   width: 10px;
}

.up   { background-position: 0 -10px }
.down    { background-position: 0 -20px }
.left    { background-position: 0 -30px }
.right   { background-position: 0 -40px }
</style>

<!-- HTML -->
<ul>
   <li class="arrow up"></li>
   <li class="arrow down"></li>
   <li class="arrow left"></li>
   <li class="arrow right"></li>
</ul>
```

The code in Listing 3.6 downloads a single image, but references four areas within this image that represent four distinct icons.

Advantages

- **Reduces HTTP-requests**

  The more images that can be combined into an image sprite, the more HTTP-requests can can be saved.

Drawbacks

- **Added complexity in development and maintenance**

  Requires making of sprites and image coordinate mapping in CSS. This can be a time consuming effort. However, there exists tools that can automate much of the process, such as Compass[18].

- **Only works for bakground images**

  Images can only be used as element backgrounds, not as inline elements (`<img>`), this can conflict with semantics if an image is considered part of the content. Sprites should therefore only be used for presentational markup (background images).

- **Harder to implement in responsive designs**

  Since sprite coordinates are explicitly stated in pixel values, they don't lend themselves to fluid sizing the way inline images do. Fluid sprites are much harder to implement. An alternative approach is to load different sized fonts under different conditions, but this would again increase maintenance complexity.

- **Breaks modularity / harder to cache**

  Since an image sprite are combine images to a single file, each image can't be cached individually. Editing a single image within the sprite would therefore require the entire sprite to be re-cached. Sprites should therefore be reserved for images that change infrequently.

- **Requires frequently paired image resources**

  The images concatenated into an image sprite should frequently be used on the same pages. If images within sprites are not used on the same pages, the performance gain of reduced HTTP-requests is lost. Sprites should therefore be reserved for frequently concurring images.

[18] `http://compass-style.org/reference/compass/helpers/sprites/`

3.2.2 Symbol fonts

Symbol fonts in computing have been around since the early nineties when Microsoft released it's Wingdings font, a collection of what is known as typography dingbats[19].

In the later years symbol fonts have seen a resurgence on the web. This is because they offer a simple way to concatenate large amounts of vectorized graphics into a single font file. Symbol font graphics are often used in UI elements such as navigation bars. Symbol fonts can be manipulated through CSS like any other font, this makes it easy to resize, change color, and add various effects (like `text-shadow`) (Suda, 2013).

Advantages

- **Multiple vector graphics stored in a single font file**

  Like image sprites, symbol fonts concatenate several graphics into a single file, reducing requests. Vectors are also infinitely scalable, which means that they can be scaled up or down in size without creating image artifacts. This is especially useful for websites with responsive designs since fonts can easily be resized for different screen sizes.

- **Easily customizable**

  Symbol fonts can use the same styling as other fonts in CSS. This means that properties like color, size and orientation are easily adaptable. Special font properties like `text-shadow` can also be applied, making symbol fonts easily customizable.

- **Works on older browsers**

  While some older browsers don't support vector formats such as SVG, vector formats for fonts have good legacy for most browsers. Browsers have the ability to choose fallback fonts if a font format is not supported (Suda, 2013, chap. 3)

[19] Ornamental characters used in typesetting. Used to denote fonts that have symbols and shapes in the positions designated for alphabetical or numeric characters.

Drawbacks

- **Must wait on CSS parsing**

  Symbol fonts need to be declared in CSS using the `@font-face` rule. As a consequence, CSS needs to be downloaded and parsed before a symbol font can be downloaded.

- **Limited color options**

  Fonts are monochromatic, meaning that a symbol font graphic can only have a single color at any one time.

3.3 Optimizing textual resources

Certain front-end optimization techniques mainly apply to textual resources such as HTML, style sheets, and JavaScript. These techniques are more general as that they do not pertain to responsive design exclusively, but should still be mentioned since overall performance optimization is particularly advantageous for mobile devices and therefore also relevant for RWD. The techniques discussed in this section will be concatenation, minification, and compression. These techniques are often used in conjunction to optimize performance.

3.3.1 Concatenation

Concatenation is the technique of combining multiple resources into a single file. This reduces the number of HTTP requests and subsequent server round trips needed to fetch multiple resources, in favor of downloading a single, larger file. Zakas (2013) notes:

> The best way to combat latency is for a Web site or application to use as few HTTP requests as possible. The overhead of creating a new request on a high-latency connection is quite high, so the fewer requests made to the Internet, the faster a page will load.

Concatenation is frequently applied to CSS and JS files, but the technique also applies to images in the form of image sprites and symbol fonts. Concatenation can be performed in a number of ways, both during production before the resources are are uploaded to the server, or dynamically on the server.

Advantages

- **Reduces HTTP-requests**

  The primary reason to use concatenation is to reduce HTTP-requests.

Drawbacks

- **Breaks modularity / harder to cache**

  As with image sprites, concatenation merges several resources, breaking their modularity. This can have a negative impact on caching. When several files are combined, editing one file requires the concatenated file to be recreated, which means that the file would need to be downloaded again to be in order to cached by a browser (Grigorik, 2013a). Individual files escape this issue, because only the modified file would need to be cached again. As with image sprites, concatenation should therefore be reserved for files that change infrequently.

- **Longer download and parse times**

  Files become longer and require more time to download. In the case of CSS and JS, this might introduce a delay in processing since a bigger file needs to be downloaded before parsing and execution can start. Concatenated textual resources and image sprites "occupy more memory on the client and require more resources to decode and process" (Grigorik, 2013a). The longer the browser has to wait to download, parse and execute style sheets and scripts, the longer it takes to render anything to the screen (as described in section 2.2.3).

3.3.2 Minification

Minification is the process of removing whitespace, comments, and characters that are not critical for execution of code. The term is also used for shortening syntax, for instance by shortening variable names in JS and rewriting CSS to use shorthand properties[20]. It is a process that can be automated by several tools, both on the server side (e.g. Google PageSpeed Module[21]) and by locally installed applications (e.g. Grunt[22], Compass).

[20] "Shorthand properties are CSS properties that let you set the values of several other CSS properties simultaneously. Using a shorthand property, a Web developer can write more concise and often more readable style sheets, saving time and energy" (Mozilla, 2013)

[21] `https://developers.google.com/speed/pagespeed/module`

[22] `http://gruntjs.com/`

Souders (2008, p. 39) states that JavaScript files sizes can be reduced as much as 20% through minification.

Advantages

- **Reduced file sizes**

  Reduces file sizes by removing segments of code not necessary for execution.

Drawbacks

- **Relatively small file size reductions**

  The reduction in file size can be relatively small. If the files are compressed (i.e. gzip) before they are downloaded from the server, the performance gain from minification can be negligible to that of the compression. Nicolaou (2013, p. 8) states:

  > Minification is beneficial for mobile, though the biggest effect is seen in comparing uncompressed sizes. If minification poses a challenge, be sure to compare the sizes only after applying gzip to get a realistic sense of the gains. They may be below 5 percent and not worth the extra serving and debugging complexity.

- **Code obfuscation**

  When all unnecessary characters, whitespace and comments are removed from the code, the legibility is reduced. This can be an issue when debugging a website without access to the original unminified resources. To prevent this problem, *source maps* have been developed, and support for these maps have been implemented in several browsers. Source maps are additional files that can be downloaded to recreate the original source code from its minified version. These maps are only loaded when debugging, and will therefore not impact performance when browsing regularly (Basu, 2013).

3.3.3 Compression

Probably the most common form of performance optimization is enabling server-side compression of textual resources. Compression is used ubiquitously because of its broad support, both for server side compression and client side (browser) decompression. The most commonly used and supported compression schemas are `gzip` and `deflate` (zlib). Both utilize a deflation algorithm based on the LZ77 (Lempel-Ziv 1977) lossless compression algorithm, which look for repeating string patterns in text. When encountering multiple occurrences of a string in a text the following occurrences of a string is replaced by a pointer to the previous identical string. This is done in the form of a pair containing the distance to the previous string, and the length of the string (Gailly and Adler, 1999). HTML, CSS, and JS code often have many repeating string patterns, and are therefore good candidates for compression (Nicolaou, 2013; Davies, 2013a). Gzip is said to reduce text file sizes by 60-80% on average, and is one of the simpler optimization techniques to implement yielding high results (Grigorik, 2013b). Fig. 3.3 illustrates a high level overview of a browser request with and without gzip compression.

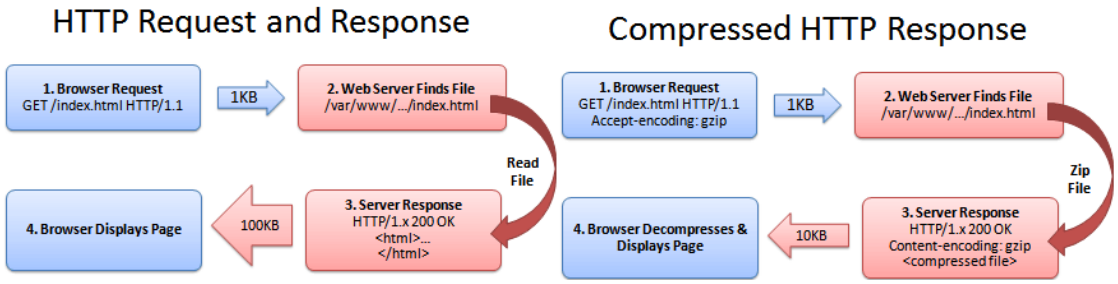


Figure 3.3: Request with, and without compression (Azad, 2007)

Yuan et al. (2005) notes that the effectiveness of compression is determined by the relation between the primary container object (CO), and the amount of embedded objects (EOs) defined within it. The CO is usually an HTML-file, while COs consists of embedded objects like images, video, audio, CSS, and JS. At the time this research was made the CO accounted for 44% of average page size, that number can be affirmed to be even less

today. As previously stated, 62% of average page loads in early 2014 consisted of image assets (The HTTP Archive, 2014), the average size and numbers of external scripts and style sheets have probably also increased since this paper was published in 2005, as it has generally done between 2010 and 2014 (The HTTP Archive, 2010, 2014). Yuan et al. state that since the CO constitute less than half of total page size, the significance of compression on page latency would be much smaller than 50%. For pre-compression the the improvement in latency for a single CO was 57,2%, whereas the number for whole page (CO with multiple EOs) was only 12,2%. The study also show that as the number of EOs increase, the improvement on page latency from compression decreases.

An improvement of 12,2% in whole page latency is still very good, considering the ease of implementation and support for compression server- and client-side. The study also shows that compression improves the "definition time" (DT) of EOs, meaning that the URLs of EOs are defined earlier, and therefore start downloading faster when a CO is compressed. Pre-compression is said to reduce DT by as much as 43,6%. This is so much that the bottleneck for page retrieval becomes the parallelism width (the number of concurrent TCP connections per domain) available. Yuan et al. based their paper on current average parallelism widths available in browsers, which was 4. As we will discuss in section 3.4.1, the average number today is 6. The paper concludes that "increasing parallelism width indeed improves page latency considerably, and compression constantly gives better performance than "No compression" in all situations" (Yuan et al., 2005). The increase in browser parallelism from 4 to 6, as well as use of techniques to increase the number of parallel downloads such as domain sharding and use of content delivery networks (CDNs), would therefore improve the effectiveness of compression on page retrieval.

The study also shows that pre-compression is always preferable to real-time compression. One should therefore always try to pre-compress as many static resources as possible. Of course, as dynamic content generation is frequent on many websites, this is not always possible.

Advantages

- **Reduces file sizes**

  Reduces the file size of textual resources like HTML, stylesheets and scripts.

Drawbacks

- **Only applicable to textual resources**

  `gzip` and `deflate` only works on textual resources. Other techniques are necessary to improve compression of images, video, and audio.

3.4 Improving server conditions

3.4.1 Domain sharding

Domain sharding can be explained as a "process of distributing page resources across multiple domains (servers), allowing browsers to open more parallel connections to download page resources" (Elkhatib et al., 2014, p. 15). This can also be described as increasing the browser parallelization. The technique is used to counter inefficiencies in HTTP, limiting the number of concurrent connections to a single domain. The HTTP 1.1 specification states that "[a] single-user client SHOULD NOT maintain more than 2 connections with any server or proxy" (W3C, 1999, section 8.1.4), meaning that when a browser is downloading resources for a web page, no more than two concurrent TCP connections should be made available per domain.

At the time HTTP 1.1 became an official standard (1999), this was considered a reasonable number. But since then, web pages have grown in both size and complexity, requiring more external resources. The hard cap on 2 connections would prove to be a limiting factor on web page performance, and because of this modern web browsers have increased their limit for parallel connections, most allowing up to 6 per domain (Souders, 2013; Grigorik, 2013a, p. 44).

According to 2013 data, the average web page downloads resources from 16 different domains. Many of these resources coming from third party content providers for functionality such as ads, widgets, and analytics. The data also shows that the average maximum of resources from a single domain is around 50 (see fig. 3.4) (Souders, 2013).

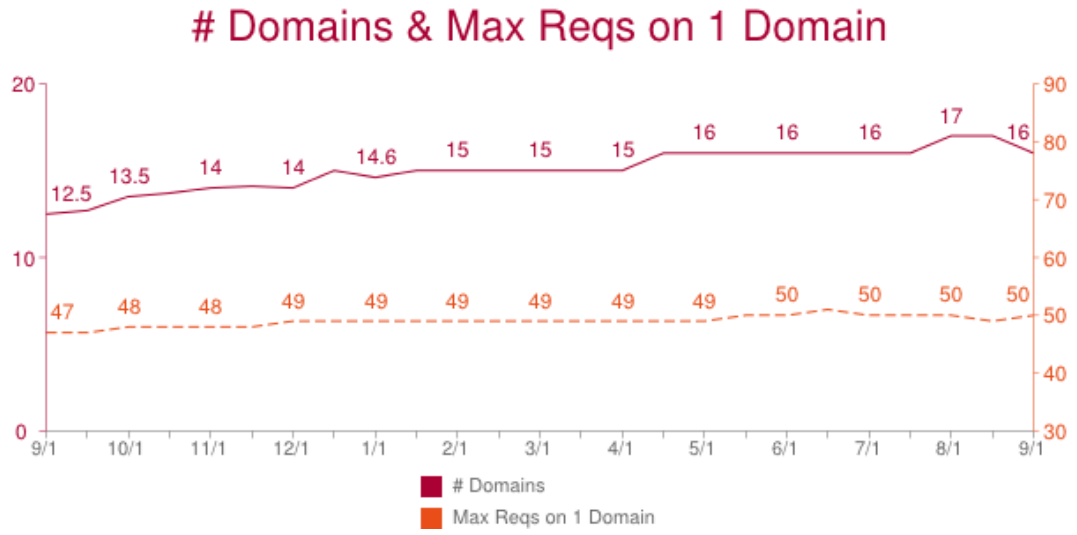


Figure 3.4: Showing the average number of domains used to request resources, and the max number of requests on a single domain from the worlds top 300K URLs in the period September 2012 - September 2013 (Souders, 2013)

Further evaluation of this data shows that the median number of resources downloaded from a single domain is 39 (Souders, 2013). With this number in mind, a browser capable of 6 parallel connections from a single domain might need upwards of 6-7 sequential requests per connection in order to download all resources from the domain. These domains are good candidates for domain sharding. This is because splitting these 39+ requests on 2 or more domains would increase request parallelization. It is important to point out that there is a trade-off here, since splitting resources across additional domains require additional time to set up new connections (DNS-resolution, three-way handshake, increasing the congestion window, etc.).

Advantages

- **Increases browser parallelization**

  For pages with a large amounts of embedded resources hosted on a single domain, domain sharding can improve download times by increasing the number of resources downloaded in parallel. Faster downloading of scripts and style sheets is also beneficial for render times, since rendering can be blocked by both.

  It is not necessary to split content across servers in order to shard a domain. Creating domain aliases (using CNAMEs) is sufficient, as the browser only limits connections to host names, not IP addresses (Souders, 2013). However, using domain aliases does not prevent additional DNS-lookups, even if resources resolve to the same host-IP. Simple content sharding is therefore relatively easy to accomplish.

Drawbacks

- **Additional DNS-lookups and connection delays**

  Domain sharding should be used with caution. Research indicates that, in most cases, resources should not be sharded across more than two domains (Souders, 2013; Chan, 2013; Mineki et al., 2013, p. 541). This is because aggressive sharding leads to extra performance costs of DNS lookups for each new domain, as well as an increased CPU and memory costs on both client and server (Souders, 2013; Grigorik, 2013a, p. 44). This can have a negative impact performance and increase the possibility of network congestion (Chan, 2013; Grigorik, 2013a, p. 44). It is important to note that different web pages on the same domain might have big disparities with regards to embedded resources. One page might request 100 images not present on other pages. This makes that the need for domain sharding difficult to asses. It is therefore important to look at the performance trade-off for different web pages across domains when evaluating the effect of domain sharding. It is also important to map which resources are frequently requested at the same time in order to determine how resources should be split into different "shards". Domain

sharding is a balancing act where performance improvement needs to be monitored across several pages. In some cases the penalty of opening additional connections might not warrant the increase in parallelization.

3.4.2 Content delivery network

One way of reducing request and response times between client and server is to reduce the distance between the two. Not only does the geographical proximity mean shorter physical travel for a network packet, but it also reduces the number of intermediary nodes on the network. One way of reducing distance between web page servers and clients is through a content delivery network (CDN). A CDN is a network of "surrogate servers" distributed around the world. The job of these surrogate servers is to cache contents from the origin server, the origin server being a website's main server. By having it's content cached at multiple locations around the world, requests for a web page can be routed to the most appropriate location. The most appropriate location is decided by *topological proximity*, which is "an abstract metric that considers a variety of criteria, such as physical distance, speed, reliability, and data transmission costs" (Vakali and Pallis, 2003). Fig. 3.5 shows how a CDN could be be distributed. CDNs are most often used to deliver static content, such as style sheets, scripts, and images (Souders, 2007, p. 20).

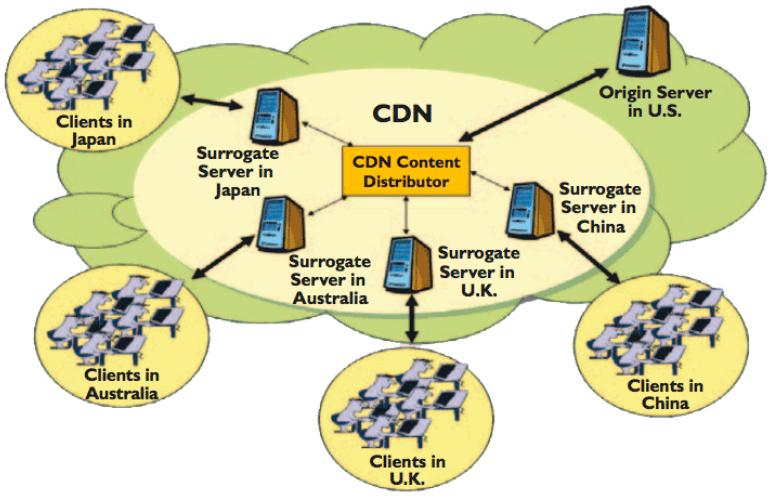


Figure 3.5: An overview of a Content Delivery Network (Pallis and Vakali, 2006)

Advantages

- **Reduces round-trip times between client and server**

  By bringing resources closer to the clients geographically, CDNs allow resources to be downloaded faster because the round-trip time between client and server is decreased.

- **Reduces load on origin server**

  Having content downloaded from a distributed network of servers can greatly reduce the load on the origin server. A CDN ensures that requests for resources are routed to the closest surrogate server instead.

- **Multiple fallback locations for resources**

  If one server becomes unavailable, requests can simply rerouted to another surrogate server. Rerouting would incur latency, but still ensure resource delivery.

- **Domain sharding**

  Distributes resources across domains, can thereby also function as a domain shard.

Drawbacks

- **Cost**

  Setting up CDNs is costly. Most websites therefore buy CDN services from a commercial provider. Using a commercial CDN provider introduces costs in storage and bandwidth usage.

- **Increased complexity for resource distribution**

  Mechanics for properly distributing resources to surrogate servers needs to be rigorous and fast. To ensure that websites are kept up to date it is therefore important that the caching mechanisms between origin and surrogate servers function properly. Using commercial CDN providers removes the ability to control this mechanism.

- **Additional DNS-lookups and connection delays**

  Using a CDN allows splitting resources across multiple domains. As with domain sharding, this necessitates additional DNS-lookups and TCP-connections for each new domain.

4 Performance tests

4.1 Test conditions

Aspects of quantitative data collection and analysis have been used in testing. Ratio data [23] has been collected in the form of different performance metrics. The test results will be analyzed and discussed in the following sections. The performance tests utilizes visual aides in the form of tables and charts. For the sake of simplicity the tables and charts have been concatenated to a single figure for each test run (see fig. 4.1).

Figure 4.1: The different elements used in test figures

The *table* collects the most important performance metrics such as load times, start render times, and kilobytes downloaded. The *waterfall chart* shows in which sequence resources are loaded, how long each resource took to download, and the constituent parts of this load time: DNS lookup, initial connection, time to first byte, and content

[23] Data measured against a quantitative scale where intervals betweens points are proportionate, and where the measurement scale has a true zero (Oates, 2005, p. 248).

download. In the desktop tests the waterfall chart also shows CPU and bandwidth utilization. The *connection chart* shows how many TCP connections were opened in order to download all resources, and how the resources were distributed across these connections. The connection chart will only be included in the figures if the number of TCP-connections differ between pre- and post-tests. Vertical lines in the waterfall and connection charts highlight important events in the web page load procedure such as *start render*, *DOM Content Loaded*, and *On Load*. The different events are marked by different colors. Fig. 4.2 and fig. 4.3 shows legends mapping events and colors. Note that there are some small differences between the mobile and desktop legends.

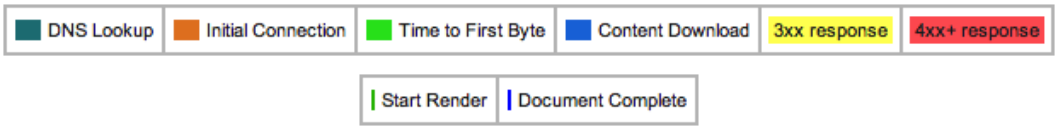


Figure 4.2: Legend for charts in the mobile tests

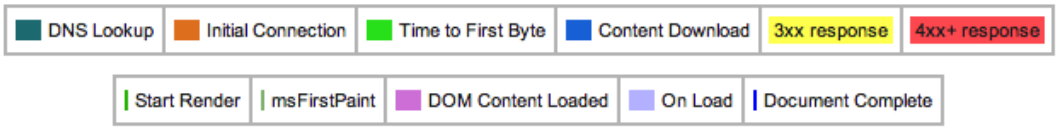


Figure 4.3: Legend for charts in the desktop tests

For each performance optimization technique, *pre-* and *post-testing* has been performed. Pre-tests measure performance before a technique has been applied, while post-testing measure performance after a technique has been applied. These kinds of 'before' and 'after' measurements are common in experiments research. The goal is to observe the test results in an attempt to identify changes attributable to manipulation or introduction of an *independent variable* (Oates, 2005, p. 131). In this case the independent variable is the performance optimization technique itself.

Each pre- and post-test has been run 5 times under mobile and desktop conditions. For each technique covered, the median runs (based on load time) under each condition will be used as the basis for further discussion. Mobile and desktop test conditions are listed

in tables 4.1, 4.2, 4.3.

Mobile

Location	**Device**	**Browser**	**Connection**	**Viewport width**
Dulles, VA, USA	iPhone 4	Mobile Safari for iOS 5.1	Shaped 3G (1,6 Mbps/768 Kbps, 300 ms RTT)	320px

Table 4.1: Specifications for mobile test conditions

Mobile v2

Location	**Device**	**Browser**	**Connection**	**Viewport width**
Dulles, VA, USA	Nexus 5	Chrome 34.0 for Android	Shaped 3G (1,6 Mbps/768 Kbps, 300ms RTT)	360px

Table 4.2: Specifications for alternative mobile test conditions (v2)

Desktop

Location	**Device**	**Browser**	**Connection**	**Viewport width**
Dulles, VA, USA	PC running Windows NT 6.1 OS	Chrome 34.0	Cable (5/1 Mbps, 28ms RTT)	1020px

Table 4.3: Specifications for desktop test instance

To ensure that rendering happens under equal circumstances in each test, style sheets have been loaded in the `<head>` element, while JavaScript is loaded at the end of the `<body>` element.

4.1.1 Notes on mobile test instances

Tables 4.1 and 4.2 show that two different mobile instances have been used in testing. The reason for this was that tests run on the iPhone 4 setup (table 4.1) failed to generate a proper connection charts. Instead of showing individual TCP-connections, only a single connection would be visible in the chart. Because of this, mobile tests in the latter part of this chapter (section 4.4 and 4.5) have been performed on a Nexus 5 setup (table 4.2) yielding more detailed charts. Network conditions are the same for both mobile instances.

4.1.2 Speed index

WebPagetest introduces a new metric for measuring performance called the Speed Index (SI) that is meant to give a more insight into performance than what is already offered by *start render* and *load* times. SI is a score which reflects how quickly web page contents are visually populated (rendered). This is done by measuring and calculating the completeness of rendering every 100 ms until full load (when the `onload` event is triggered) ((WebPagetest, 2014)). Figure 4.4 illustrates how the speed index score is calculated. As we see from the figure, example A has a higher percentage of visual completeness at an earlier stage than does example B. The result is a lower SI score for A than for B. The lower the score, the faster the page is at reaching complete page rendering. It is worth noting that SI scores are based on processing video frames of page rendering at fixed time intervals, results are therefore not totally reliable for all test situations.

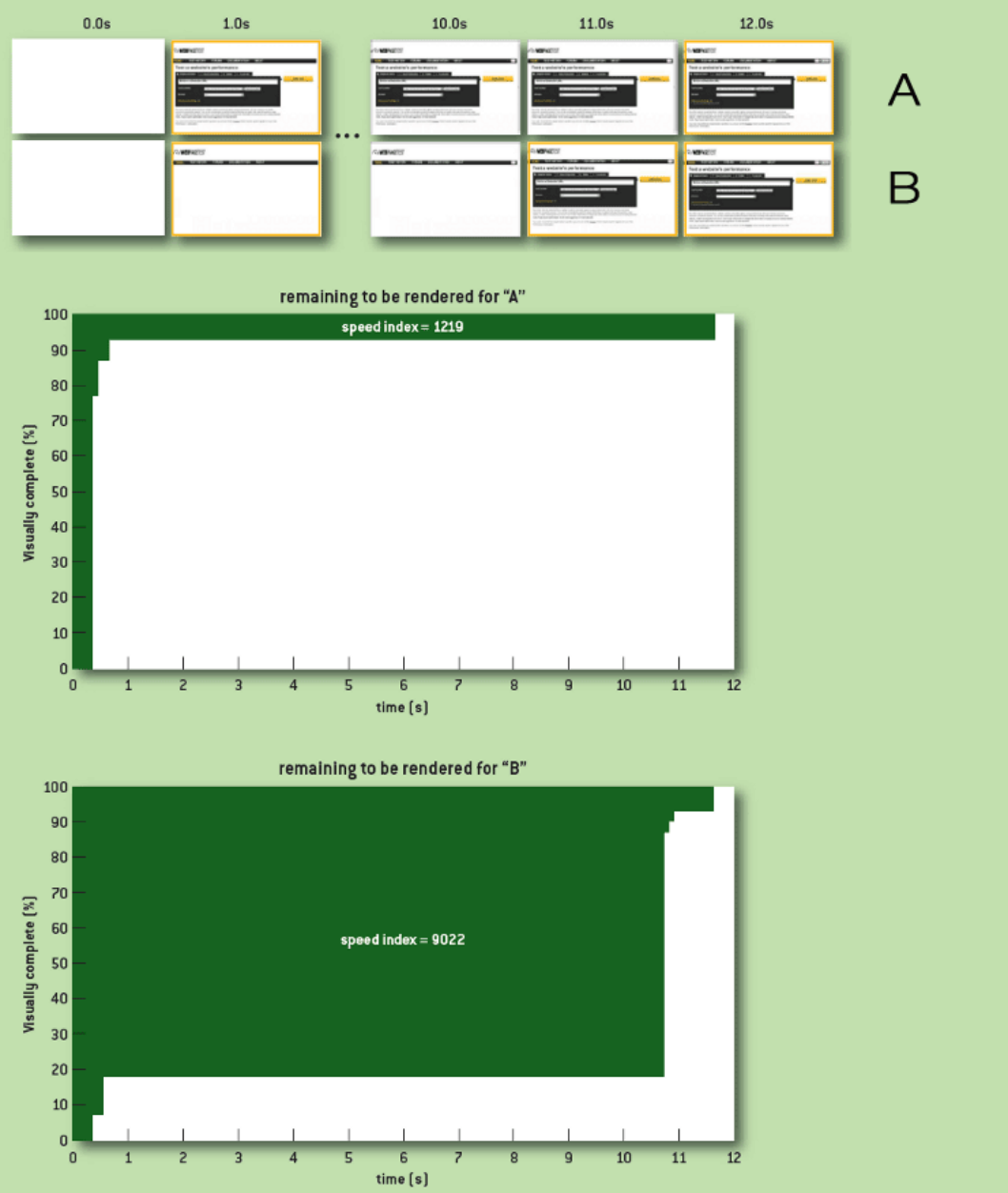


Figure 4.4: How the Speed Index score is measured (Meenan, 2013)

4.2 Implementing responsive images

For the tests in this section, a page containing 3 images was loaded. Each image exists in four different sizes: extra small, small, medium, and large. Breakpoints and corresponding image sizes for techniques utilizing media queries is shown in table 4.4.

Image size name	**Image width (px)**	**Targeted screen width (px)**
Extra small	480	< 480
Small	768	480 – 768
Medium	992	768 – 992
Large	1200	> 992

Table 4.4: Breakpoints used for images

4.2.1 Picturefill

Results (mobile)

When testing Picturefill the results showed a predictable pattern, namely that performance was improved on mobile browsers. Fig. 4.5 and fig. 4.6 shows the results of the pre- and post-tests.

The post-test shows that total load times is improved by 4,019 seconds when using Picturefill. The decrease in load time can in this case be attributed to the fact that Picturefill ensures that more appropriately sized images are loaded. In this case the "extra small" images (480px wide) are loaded, while the pre-test test downloaded the "large" images (1200px wide). This is reflected in the total size of image resources downloaded, which are reduced from 996KB to 202KB, an 80% decrease.

It is interesting to note the results of start render times. Even though Picturefill downloads smaller images on mobile, it does not improve render times in this test. In fact, rendering starts approximately at the same time as the control test(the 18 ms difference

is negligible), and both test are "visually complete" at the same time (6 seconds). This is because the browser is not dependent on images to start rendering, therefore it can start as soon as the CSS is processed.

The waterfall chart in fig. 4.6 shows one of Picturefill's main disadvantages, namely image downloads are dependent on the `picturefill.js` script. In the case of Picturefill, the image paths themselves are embedded by `picturefill.js`, therefore the pre-parser has no ability to start fetching the images before the script has been executed. As we can see from the waterfall chart in fig. 4.6 this actually results in the images being fetched last, almost 6 seconds after the initial HTML request. In the pre-test the images start to download after only 1 second. In the post-test, the images are the last resources to be downloaded. Contrast this to the pre-test, where the images are downloaded in parallel with most of the other resources. From the waterfall graph in fig. 4.6 we observe that it takes 4,547 seconds for `picturefill.js` to be downloaded, parsed and executed before the browser can start loading any images.

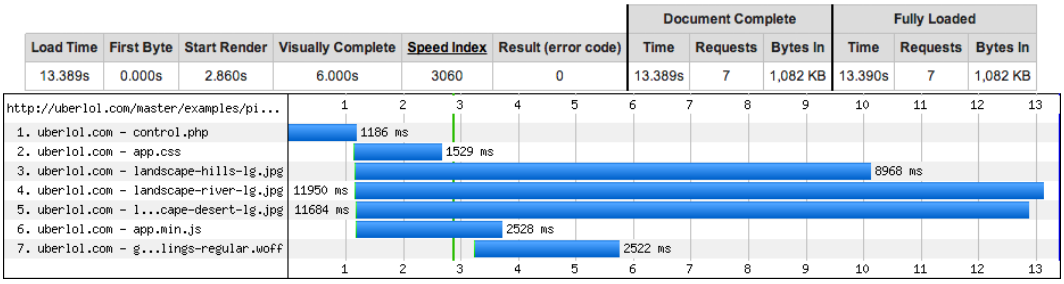


Figure 4.5: Mobile – Pre-test without Picturefill

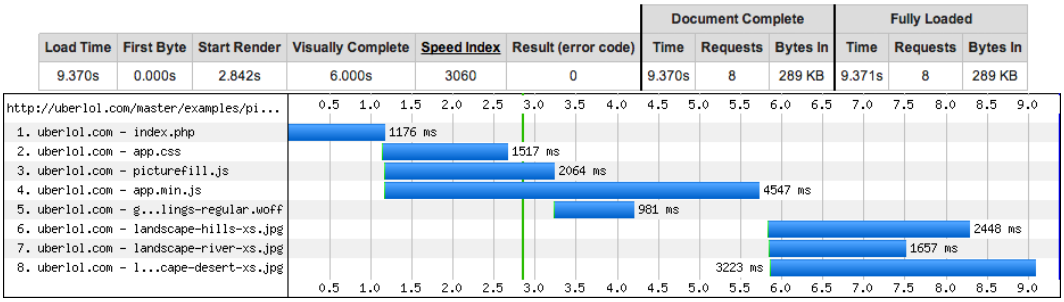


Figure 4.6: Mobile – Post-test using Picturefill

Results (desktop)

The results from the mobile tests gave indications to the effect of Picturefill with regards to download speed, order, and render times. The same effects were observed when loading the page on a larger screen with higher bandwidth. Fig. 4.7 and fig. 4.8 shows the result from the pre- and post-tests respectively.

The waterfall charts show the same shapes in the desktop tests as in the mobile tests. Since the viewport is bigger in the desktop tests, the Picturefill post-test now downloads the "large" images, the same as in the pre-test.

As we could expect under these circumstances, the time to load all content is prolonged by the presence of `picturefill.js` (fig. 4.8). Images start to download approximately 1 second later than in the pre-test, and total load time is 0,397 seconds longer. However, rendering is shown to start earlier in the post-test utilizing Picturefill. This can perhaps be attributed to the fact that Picturefill prevents the images from being downloaded in parallel with the other resources, allowing the main CSS (`app.css`) file which blocks rendering to be downloaded slightly faster.

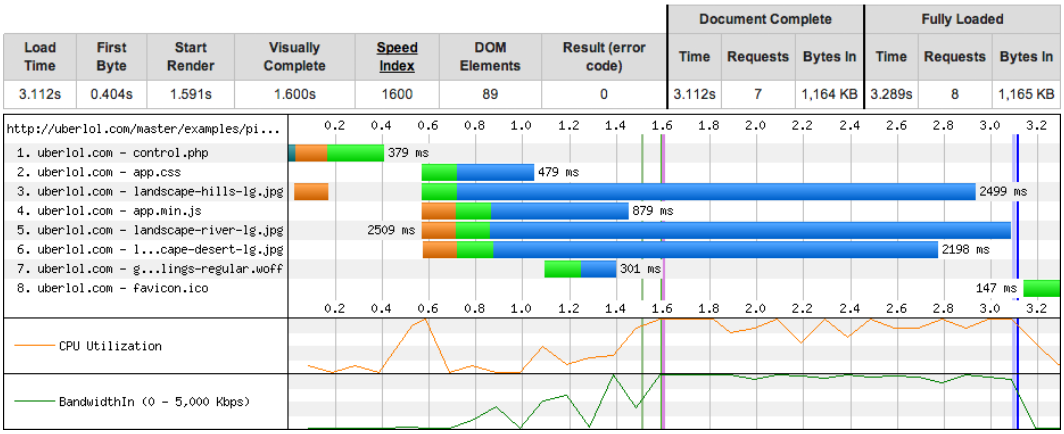


Figure 4.7: Desktop – Pre-test without Picturefill

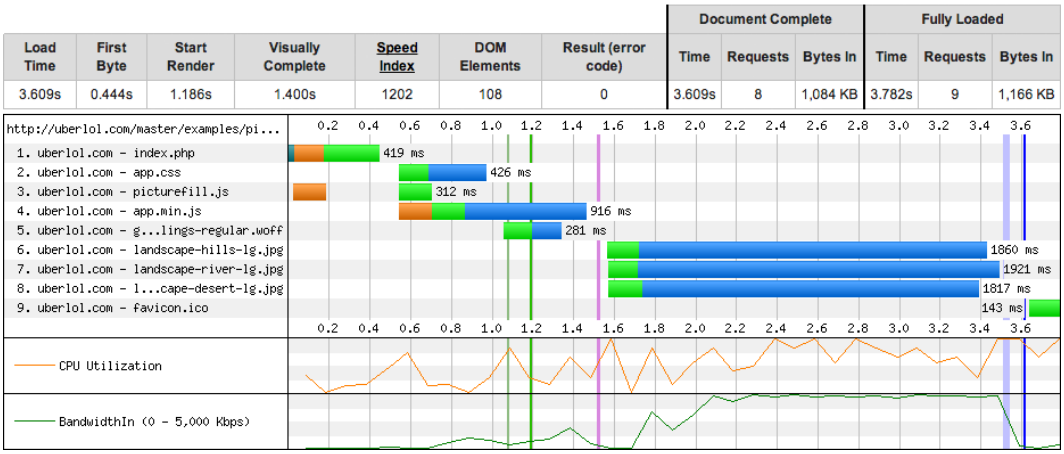


Figure 4.8: Desktop – Post-test using Picturefill

Conclusion

The results show that the using a javascript polyfill solution to responsive images can yield positive results. The impact is largest on mobile browsers, where the download of more appropriately sized images yield savings in both load times and sizes. Picturefill does not notably affect render times, however, the script significantly delays the start time for image downloads, which can result in longer total load times.

The fact that image downloads start later can be problematic when images appear in the visible part of a web page[24] when the page starts to render. In fig. 4.6 we observe that page rendering starts 3 seconds before the images even start downloading. This leaves blank areas in the layout where the images are later loaded. This introduces the same problem with content reflow as discussed in section 3.1.4. It could therefore be necessary to implement a "padding-bottom hack" to reserve space for images loaded with Picturefill as well.

[24] Also known as "above the fold" content.

4.2.2 Adaptive images

In the AI test only the large images (1200px wide) are embedded in the HTML. This is because AI automatically resizes images. It is therefore not necessary to manually create different image sizes prior to testing.

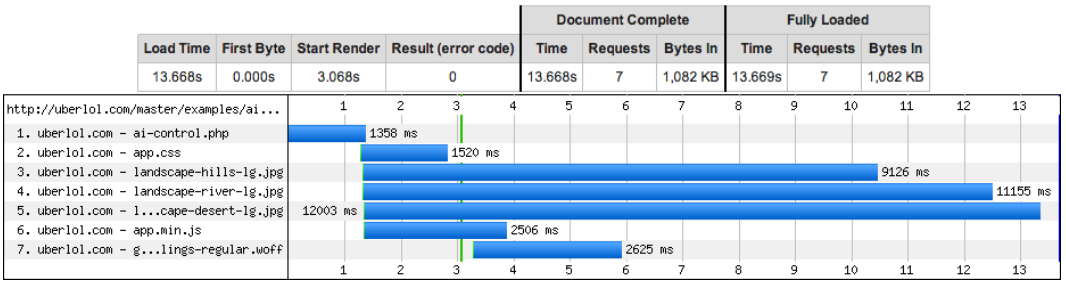
Results (mobile)

When testing the effect of AI, it was necessary to see how the technique performed both with an existing cache of the images at the correct sizes, and with an empty image cache. This was done to see how big the impact was on response time when AI had to rescale images on the fly. Fig. 4.10 shows the result of AI on a mobile browser (320px wide viewport) with an empty image cache, while fig. 4.11 shows the result of subsequent loads when the images already existed in the cache.

As shown in the waterfall graphs, the time to download the `index.php` file and images is longer on the first view (empty cache, fig. 4.10) than on the subsequent view (fig. 4.11). This can be explained by the fact that AI needs to perform image scaling and substitution on the server before the images can be sent to the client. From the graphs we see that this does not have a big impact on start render times (which are only dependent on the style sheet), but the overall download times for the images is extended by 1-2 seconds on an empty cache.

Fig. 4.9 shows the results from the pre-test where AI was not used. As a result of less processing instructions we can see that `index.php` is faster to download than the two other tests, and therefore 300-500 ms faster to start rendering (witnessed by the vertical green line in the waterfall chart). However, since the pre-test does not have the ability to scale down images, it has to download the original image resources. At this resolution that means an additional 907KB compared to the AI post-tests (1082KB instead of 175KB). As expected, this adds considerably to the download time for the images, adding several additional seconds to download each image. AI with a primed

cache therefore ends up loading all images 4,807 seconds faster than the pre-test. Even with an empty cache, AI loads 3,572 seconds faster than the pre-test.

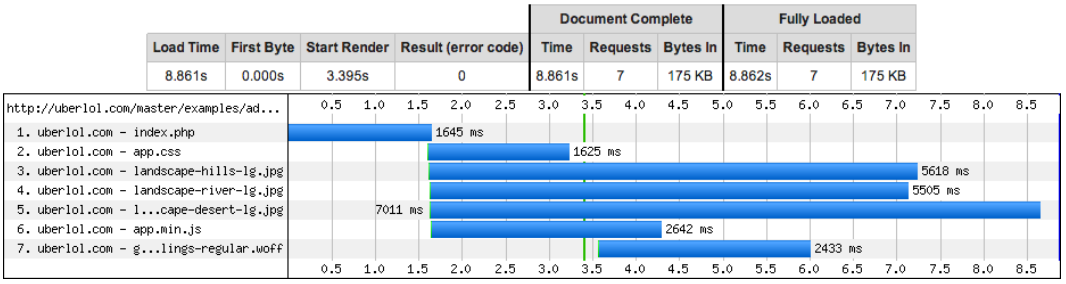


				Document Complete			Fully Loaded		
Load Time	First Byte	Start Render	Result (error code)	Time	Requests	Bytes In	Time	Requests	Bytes In
13.668s	0.000s	3.068s	0	13.668s	7	1,082 KB	13.669s	7	1,082 KB

Figure 4.9: Mobile – Pre-test without AI

						Document Complete			Fully Loaded		
Load Time	First Byte	Start Render	Visually Complete	Speed Index	Result (error code)	Time	Requests	Bytes In	Time	Requests	Bytes In
10.096s	0.000s	3.555s	9.000s	4160	0	10.096s	7	175 KB	10.097s	7	175 KB

Figure 4.10: Mobile – Post-test using AI (empty cache)



				Document Complete			Fully Loaded		
Load Time	First Byte	Start Render	Result (error code)	Time	Requests	Bytes In	Time	Requests	Bytes In
8.861s	0.000s	3.395s	0	8.861s	7	175 KB	8.862s	7	175 KB

Figure 4.11: Mobile: Post-test using AI (full cache)

Results (desktop)

In the desktop tests, both pre- and post tests download the original images. This is because the screen-width is wider than the largest applicable breakpoint (992px) set

in AI. This allows us to see what effects AI has when no image downscaling is needed, observing if the AI script adds to additional load/render times, or if it degrades gracefully.

As observed in fig. 4.12 and fig. 4.13, AI seems to have little negative impact on performance under these conditions. Load times as well as start render times are similar in both tests. The AI post-test shows slightly longer download times for the images (approximately 200 ms on average) than the pre-test. This might be linked to the processing time for the AI script file.

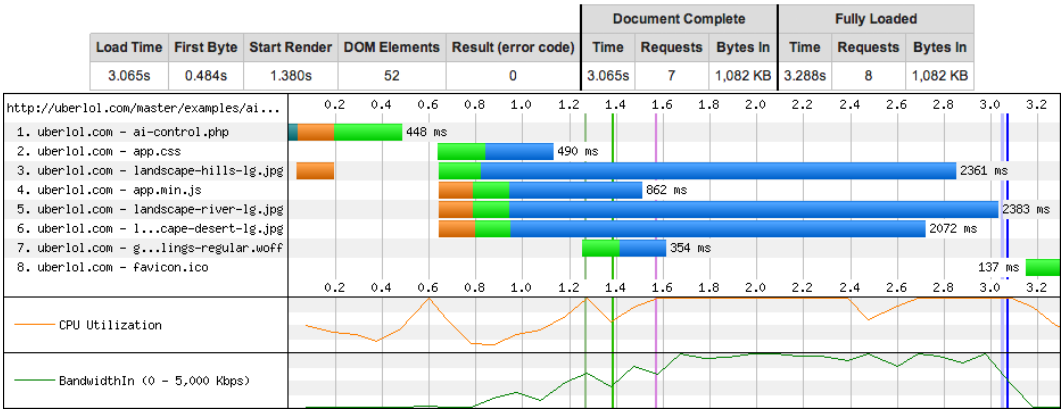


Figure 4.12: Desktop – Pre-test without AI

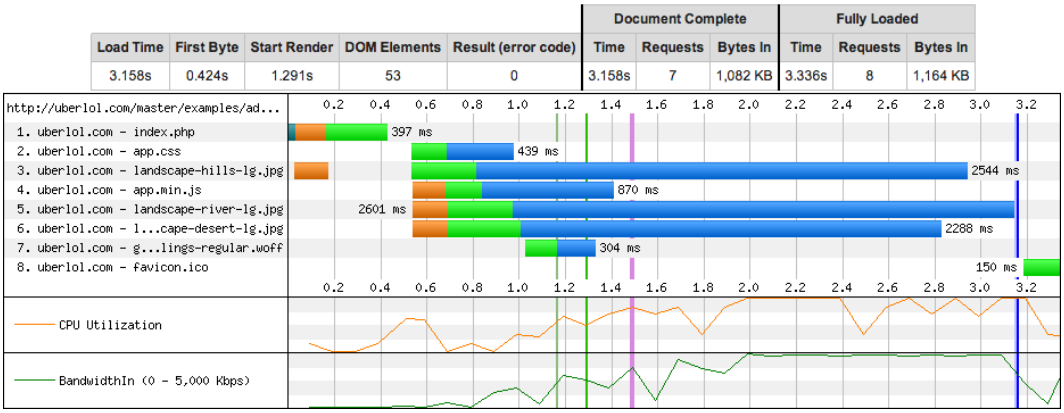


Figure 4.13: Desktop – Post-test using AI

Conclusion

Adaptive images is a solution that is easy to implement and that automates much of the work of adapting image sizes to different display sizes. The fact that most of the work is done automatically server-side is one of this techniques main advantages, saving time and reducing complexity of development and maintenance. AI also separates out all the logic for selecting appropriate images from both HTML and CSS. The tests show that there is some inherent processing overhead associated with AI, but that in most cases this is ameliorated by the improvements it offers in load times and reduction in load sizes. Significant improvements over the pre-test was observed when AI was used under mobile conditions. The post-tests show that AI has more processing overhead when scaling images on the fly, and that AI operates more efficiently once the images have been cached. Lastly we saw that AI might have some minor performance drawbacks when serving images at original sizes, but given the improvements it offered under mobile conditions, this seems like an affordable compromise.

4.2.3 Clown car technique

Results (mobile)

The tests show that CCT has the same problem with image loading delay as Picturefill. In the case of CCT, image downloads are dependent on corresponding SVG files. This delays images from downloading since the SVG files need to be downloaded first. Fig. 4.15 shows that it takes the first SVG file (request number 4) 3,432 seconds to download and parse before it can start downloading its corresponding image resource (request number 8). In this case it actually takes longer to download and parse the SVG file than it does to download the image itself. This is interesting since the size of the SVG file (564 bytes) is much smaller than the corresponding image downloaded (65KB). However, since CCT enables the download of smaller sized images for the given viewport-size, it still performs better than the pre-test (fig. 4.14). Since the small images downloaded in the post-test load faster than the large images in the pre-test, total load time is reduced

by 4,748 seconds. The smaller images download almost three times as fast, and reduces total load size by 773KB.

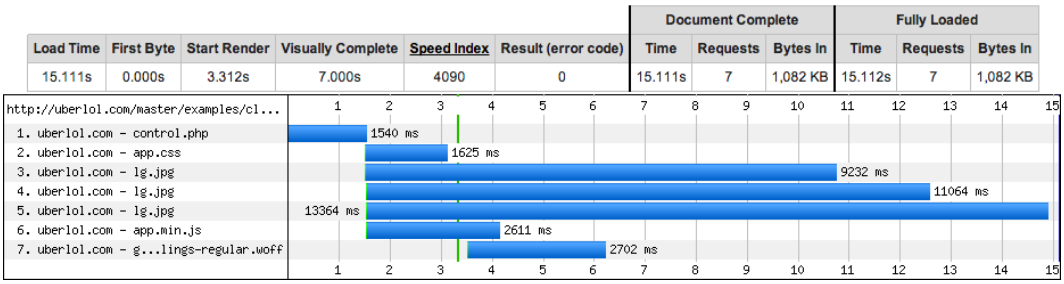


Figure 4.14: Mobile – Pre-test without CCT

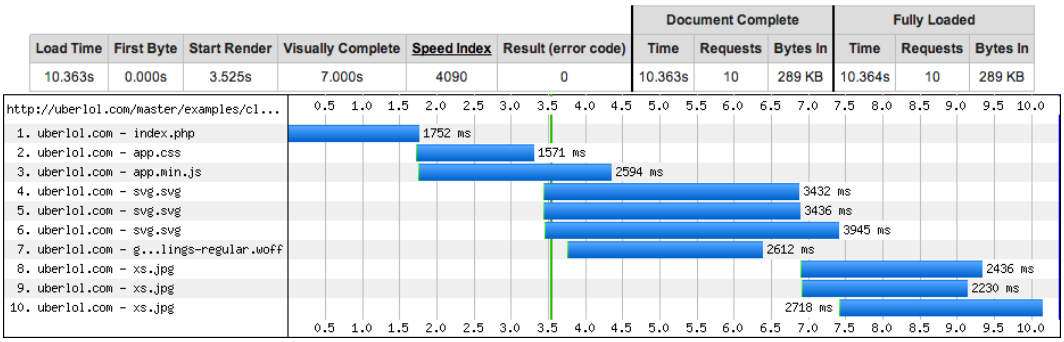


Figure 4.15: Mobile – Post-test using CCT

Results (desktop)

The desktop tests are based on a viewport measured at 1020px wide. Picturefill used this width as the basis for its media queries, while AI used the absolute screen-width, which is actually 1920px wide. Both Picturefill and AI triggered downloading of the largest images (1200px wide) in the desktop post-tests. However, because of the test web page's layout, the parent element of the images never stretches beyond 970px wide. Therefore only the medium sized image (992px wide) would be necessary to fill the parent element. Since the breakpoints in the SVG files are relative to the 970px-wide parent element, the medium-sized image specified by the breakpoint range between 768-992px is loaded in the desktop post-test (fig. 4.17).

As observed from fig. 4.17, loading smaller images saves the post-test 197KB over the pre-test (fig. 4.16).However, the post-test still performs poorer than the pre-test, both with regards to load and render times. This is largely due to the fact that the SVG files considerably delay the download of the images. Unlike in the mobile test, the time to download and parse the SVG files is much shorter than downloading the images. Still it takes roughly 1,7 seconds to start downloading the first image in fig. 4.17, while it only takes 0,8 seconds in fig. 4.16. Another consequence of the additional HTTP-requests introduces by the SVG files is lower bandwidth utilization. In fig. 4.16 we see that the pre-test reaches peak bandwidth speeds after 1,8 seconds, which it sustains for additional 2,2 seconds. In contrast the post-test (fig. 4.17) reaches peak speed after ~2,7 seconds, which is only sustained for additional 0,7 seconds. The end result is that CCT degrades performance, despite loading smaller images.

Load Time	First Byte	Start Render	Visually Complete	Speed Index	DOM Elements	Result (error code)	Document Complete			Fully Loaded		
							Time	Requests	Bytes In	Time	Requests	Bytes In
3.064s	0.528s	1.287s	3.100s	1376	86	0	3.064s	7	1,082 KB	3.223s	8	1,165 KB

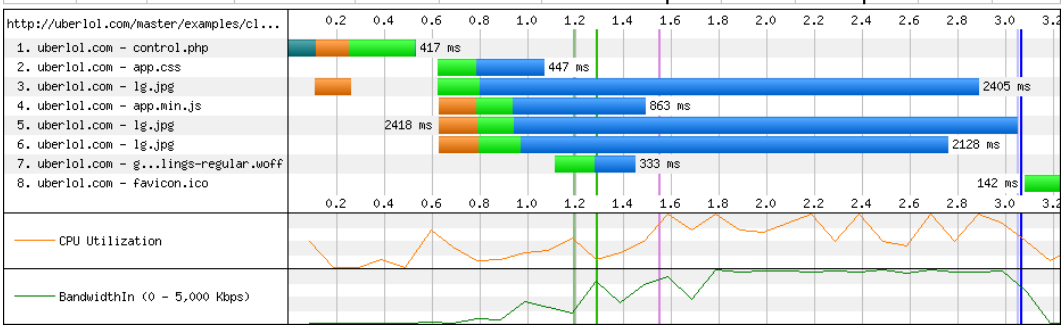


Figure 4.16: Desktop – Pre-test without CCT

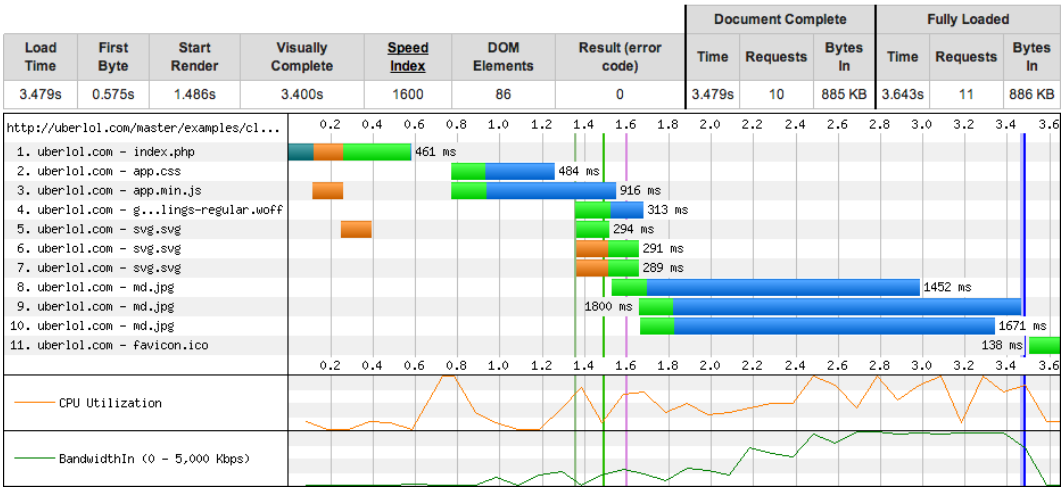


Figure 4.17: Desktop – Post-test using CCT

Conclusion

The use of CCT enables devices with smaller screens to download smaller sized images, which reduces the total load size of a web page. However, since CCT requires an additional SVG file to be downloaded for each image, the total number of requests is increased. More requests leads to more round trips between server and client, which negatively impacts performance. CCT therefore degrades performance on larger screens by requiring redundant HTTP-requests. CCT is also far more complex to implement than the other responsive image solutions examined in this thesis. The combination of added complexity and degraded performance under desktop conditions together makes CCT the least applicable solution to responsive images tested in this thesis

4.2.4 Lazy loading

As discussed in section 3.1.4, many JavaScript solutions exist to "lazily" load images once they appear in, or near, the viewport area. For our test case we wanted to examine if it was possible to combine lazy loading with responsive image solutions. As it turns out, this is not too complicated to achieve. In the following test some minor modifications of

Picturefill and addition of a few JavaScript functions allowed it to lazy load images. This was done by replacing the `data-picture` attribute (necessary to trigger Picturefill) with a `lazyload` class on the image container element. The modified script would then monitor if an element with the class `lazyload` was scrolled into the browser viewport. If so, the script would reapply the `data-picture` attribute, triggering the execution of the original Picturefill script and loading an appropriately sized image.

Since these tests only run a slightly modified version of Picturefill we will use the test results from section 4.2.1 to compare results.

Results (mobile)

From the waterfall graph in fig. 4.18 we can see that no images where loaded in this test. This is because the image-containing element never entered the viewport on initial page load. As a result we see that that this test (fig. 4.6) loads 6,633 seconds faster than the pre-test in fig. 4.5 and 2,614 seconds faster than the post-test in fig. 4.6.

Through additional manual testing it was confirmed that the page used for these post-tests would in fact lazy load images as they were scrolled within the viewport.

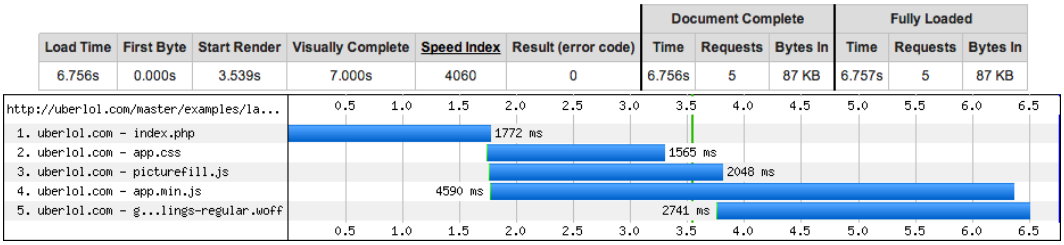


Figure 4.18: Mobile – Post-test lazy loading images

Results (desktop)

Fig. 4.19 shows the results of lazy loading Picturefill in a desktop browser. Since images still appeared outside the viewport on initial load, they are not loaded in this test either. As a result, loading the page is considerably faster than in fig. 4.7 and fig. 4.8.

Load Time	First Byte	Start Render	Visually Complete	Speed Index	DOM Elements	Result (error code)	Document Complete			Fully Loaded		
							Time	Requests	Bytes In	Time	Requests	Bytes In
1.884s	0.591s	1.588s	2.000s	1604	101	0	1.884s	5	91 KB	2.074s	6	170 KB

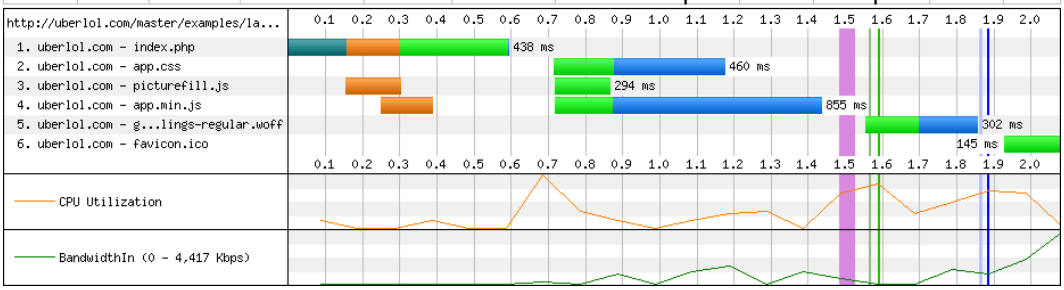


Figure 4.19: Desktop – Post-test lazy loading images

Conclusion

As long as the implementation of lazy loading on a site can overcome the challenge of browser reflows, for instance by using the "padding-bottom hack" or similar solutions (like "Responsive image placeholders"(Hinchliffe, 2013)), it can yield large improvements on performance. Initial load times are shortened because only images within the viewport are loaded, and only the images that the user actually views are downloaded. This save a lot of bandwidth for users who do not scroll through the entire page, which could be particularly beneficial for users on mobile devices with restricted data plans.

4.3 Optimizing UI graphics

4.3.1 Image sprite and symbol font

Since image sprites and symbol fonts often share the same use cases, it was natural to do a combined test of the two techniques. Both techniques are used to concatenate decorative graphics into a single file. For these tests, the pre-test loads each graphic individually as inlined images in HTML, while the post tests load them as background images in CSS. The first post-test uses an image sprite, while the second uses a symbol font. The combined size of the of the individual images are 10,2KB, the size of the image sprite is 4,8KB, and the symbol font 3,7KB. In each of the tests the icons have been scaled at 32 by 32 pixels.

Results (mobile)

In the mobile tests, the post-test using a symbol font (fig. 4.22) performed slightly better than the image sprite (fig. 4.21) and the pre-test using individual images (fig. 4.20). The difference between symbol font and image sprite is insubstantial. Both with regards to load time and start render time, the difference is less than 70 ms. However, the time to download the image sprite is longer than the symbol font. In fig. 4.21 the sprite (`icon-s5e8ef1797d.png`) takes 2,328 seconds to download, while the symbol font (`icomoon.woff`) is in fig. 4.22 observed to only take 0,961 seconds to download.

The pre-test using individual images (fig. 4.20) spends more time downloading resources, observed by a load time more 1,5 seconds slower than both the post-tests. The aggregate time to download all images is 5,415 seconds. The longer total load can be explained by the 19 additional requests needed to download all images, adding extra server round trips. However, the pre-test shows one advantage of loading the images individually. Since images are inlined in HTML, they can be requested at the same time as other resources. Both image sprites and symbol-fonts are style sheet dependent, and therefore have to wait until the CSS file has been downloaded and parsed before they can start to

download.

						Document Complete			Fully Loaded		
Load Time	First Byte	Start Render	Visually Complete	Speed Index	Result (error code)	Time	Requests	Bytes In	Time	Requests	Bytes In
7.564s	0.000s	3.808s	4.600s	2752	0	7.564s	25	100 KB	7.565s	25	100 KB

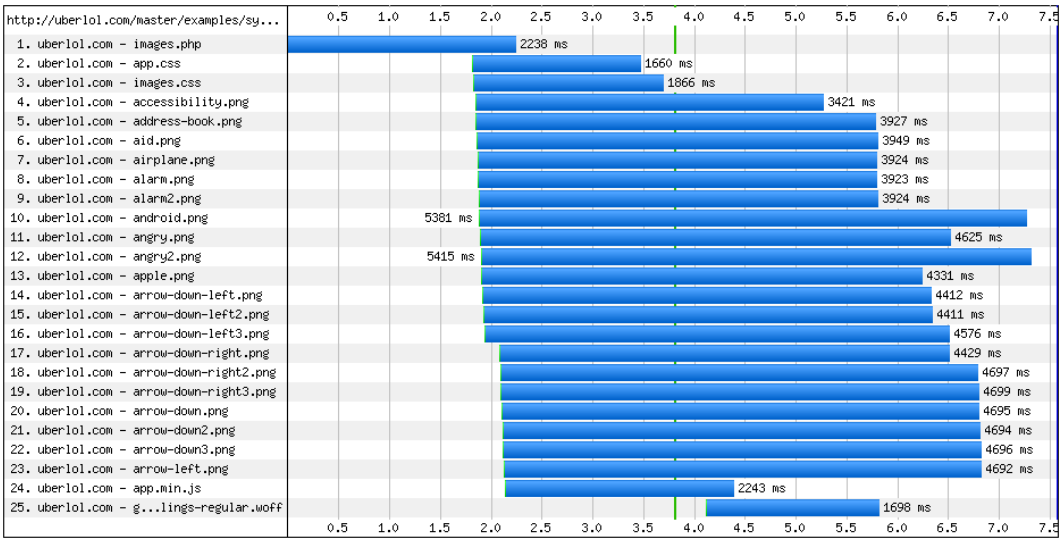


Figure 4.20: Mobile – Pre-test test using individual images

						Document Complete			Fully Loaded		
Load Time	First Byte	Start Render	Visually Complete	Speed Index	Result (error code)	Time	Requests	Bytes In	Time	Requests	Bytes In
6.001s	0.000s	3.535s	7.000s	4600	0	6.001s	6	90 KB	6.002s	6	90 KB

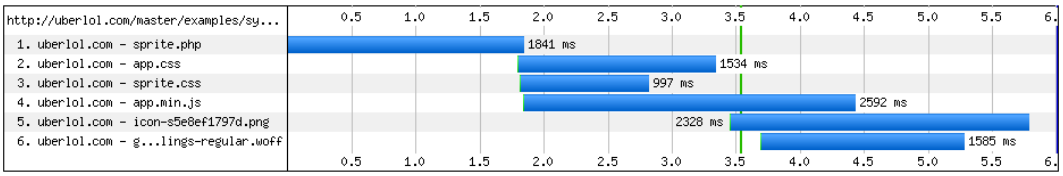


Figure 4.21: Mobile – Post-test test using image sprite

						Document Complete			Fully Loaded		
Load Time	First Byte	Start Render	Visually Complete	Speed Index	Result (error code)	Time	Requests	Bytes In	Time	Requests	Bytes In
5.957s	0.000s	3.598s	6.000s	4400	0	5.957s	6	89 KB	5.958s	6	89 KB

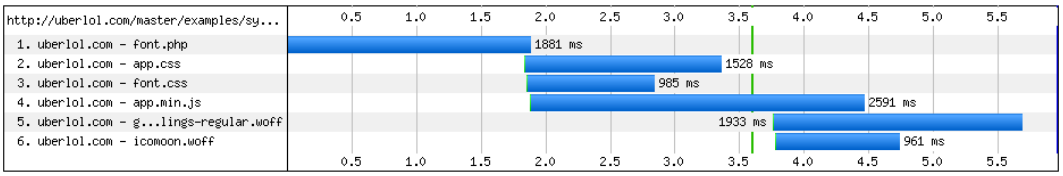


Figure 4.22: Mobile – Post-test using symbol font

Results (desktop)

The desktop browser tests show similar load and render times for all three tests. Under these circumstances the post-test using symbol font (fig. 4.25) is actually the slowest. The differences are pretty small. Load time for the pre-test is 1,818 seconds (fig. 4.23); for the symbol font post-test, 1,884 seconds; and for the image sprite post-test 1,790 seconds (fig. 4.24). The fact that RTTs are much lower on the cabled connection, 28 ms as opposed to 300 ms, can explain why the difference between pre- and post-tests are smaller than under mobile conditions. Even though total load times are similar under desktop conditions it still takes longer to download all individual images than the image sprite and symbol font. Fig. 4.23 shows that it takes 0,820 seconds from the first image starts downloading, until the last image is finished downloading. In contrast it only takes 0,264 seconds to download the image sprite (fig. 4.24), and 0,143 seconds to download the symbol font (fig. 4.25)

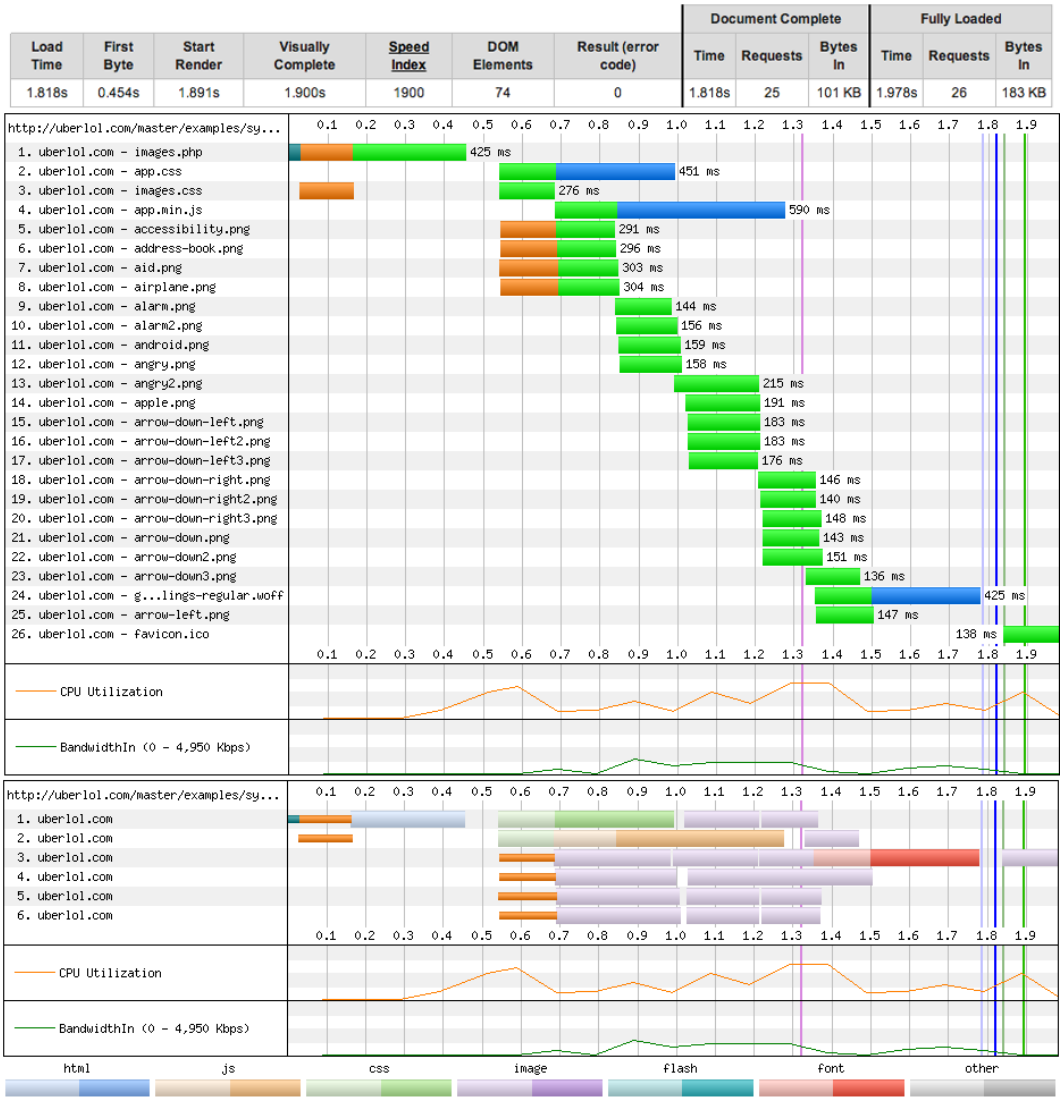


Figure 4.23: Desktop – Pre-test test using individual images

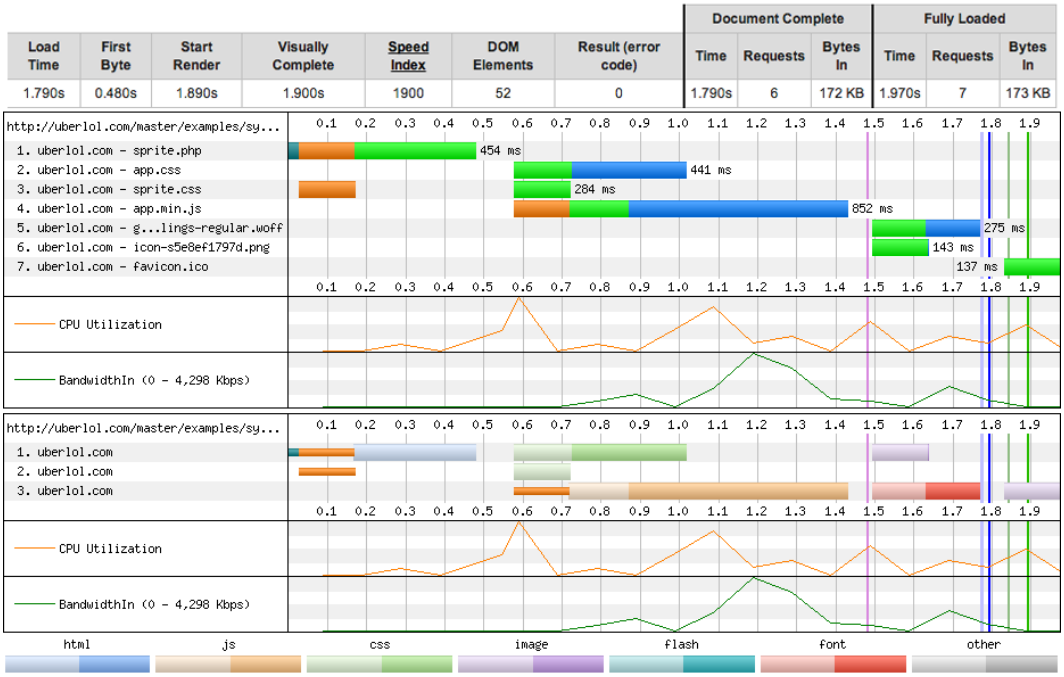


Figure 4.24: Desktop – Post-test test using image sprite

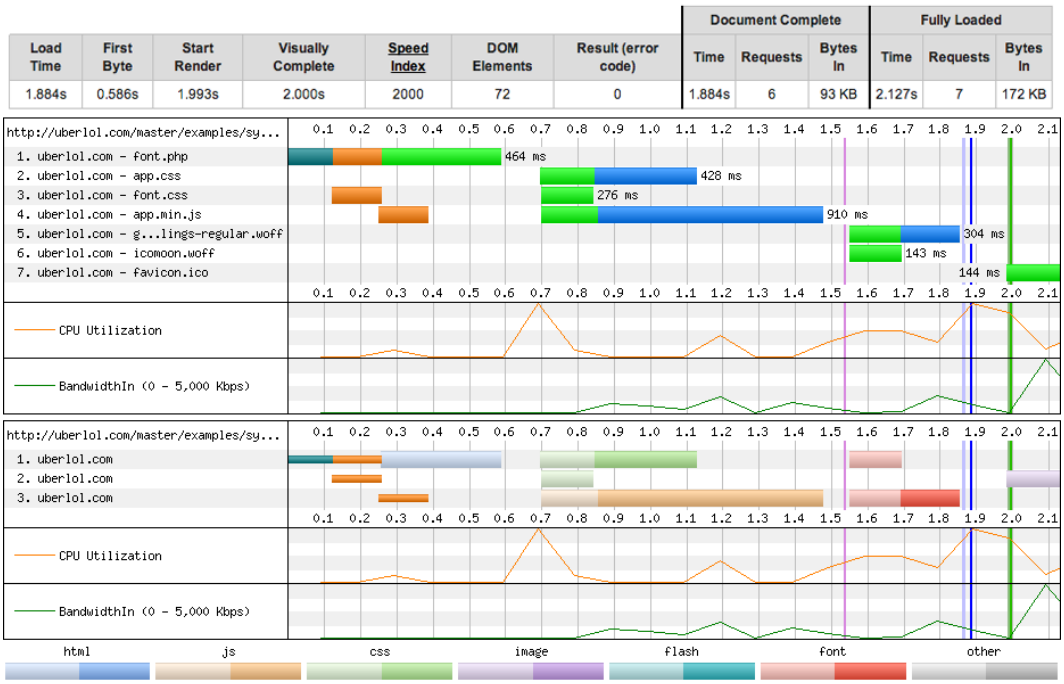


Figure 4.25: Desktop – Post-test using symbol font

Conclusion

Usage of symbol fonts and image sprites over individual images has a higher benefit on mobile networks where RTTs are longer than on cabled networks. Inlining individual images in HTML allows the browser to start downloading them earlier than if they are loaded from a sprite or font. However, if the images are meant for decorative purposes, inlining them in HTML breaks the separation between presentation and content. The fact that symbol fonts are vector-based makes them especially suitable for responsive design. This is because vector-based graphics are infinitely scalable. Symbol fonts are also easily customizable through CSS. Image sprites are less flexible, but suit certain types of images better, such as multicolored graphics or photographs.

4.4 Optimizing textual resources

When testing techniques for optimizing textual resources, it was decided to run a series of combinations of different techniques. This way it would be easier to see how the tests affected each other, as well as their relative contribution to performance. The techniques covered in this section are minification, concatenation, and compression. Minification and concatenation have only been performed on CSS and JavaScript files, while compression acts on all textual resources, including HTML. 3 CSS files and 4 JavaScript files were used in the tests. When measuring concatenation these files were combined to a single CSS file, and a single JavaScript file.

4 tests have been run under both on mobile and desktop conditions, first using server-side compression, then without compression. The four tests were:

1. No minification or concatenation (pre-test)
2. Minification only (min)
3. Concatenation only (concat)
4. Both minification and concatenation combined (min + concat)

This makes up a total of 16 tests (4 tests on mobile + 4 tests on desktop, with and without compression).

4.4.1 Minification and concatenation with compression

This section looks at the effect of minification and concatenation when server-side compression is enabled. The section discusses four tests. The first applying neither minification nor concatenation, the second applying only minification, the third applying only concatenation, and the fourth applying both minification and concatenation.

Results (mobile v2)

Technique	**Start render (s)**	**Diff. pre-test (%)**	**Load time (s)**	**Diff. pre-test (%)**	**Total load size (KB)**
Pre-test	8,422	–	14,497	–	302
Min	6,922	17,81	8,896	38,64	165
Concat	7,842	6,89	10,325	28,78	299
Min + concat	6,332	24,82	6,528	54,97	164

Table 4.5: Mobile v2 tests with compression

The pre-test (fig. 4.26) shows the results when neither minification nor concatenation is utilized. Load time over 3G-connection is 14,497 seconds, while first render happens after 8,442 seconds. Time to download all style sheets and scripts is 11,852 seconds.

When using minification (fig. 4.27), the load time is improved by 5,601 seconds and render time is improved by 1,450 seconds. Minification in this case reduces overall load size by 45,4% (302 KB to 165 KB). Time to download all style sheets and scripts is 4,751 seconds.

The third test (fig. 4.28) shows that concatenation alone yield similar improvements to minification. The total nuber of requests is reduced from 9 to 4, which results in the

browser only utilizing 2 TCP-connections instead of 6. Load time is reduced by 4,172 seconds from the pre-test, while start render is improved by 0,580 seconds. Time to download all style sheets and scripts is 6,076 seconds.

The fourth test (fig. 4.29) shows the result of combining both minification and concatenation. The results indicate that using both techniques in tandem has even better results than using each individually. This could be expected, since the techniques have a different grounds for improving performance (reduction of file sizes vs. reduction of requests). Load time is improved by 7,969 seconds, while start render time is improved by 2,100 seconds. Time to download all style sheets and scripts is 4,162 seconds.

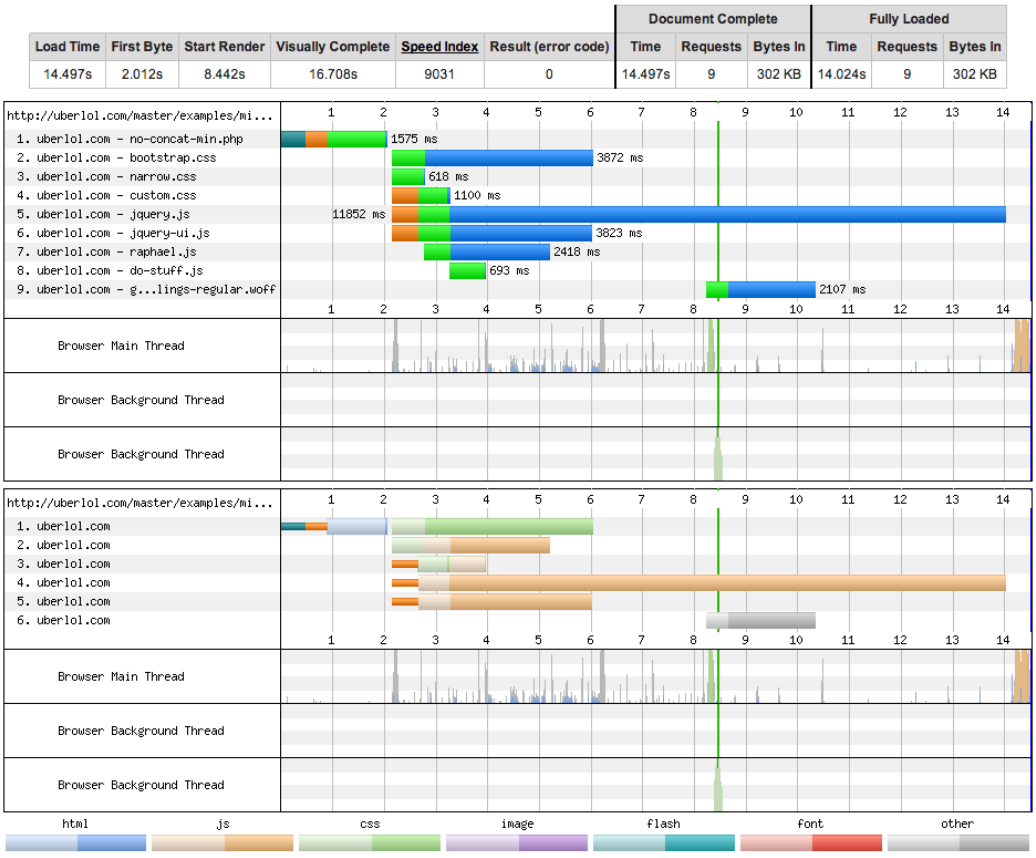


Figure 4.26: Mobile v2 – Pre-test with no minification or concatenation

						Document Complete			Fully Loaded		
Load Time	First Byte	Start Render	Visually Complete	Speed Index	Result (error code)	Time	Requests	Bytes In	Time	Requests	Bytes In
8.896s	1.532s	6.922s	9.138s	7079	0	8.896s	9	165 KB	8.888s	9	165 KB

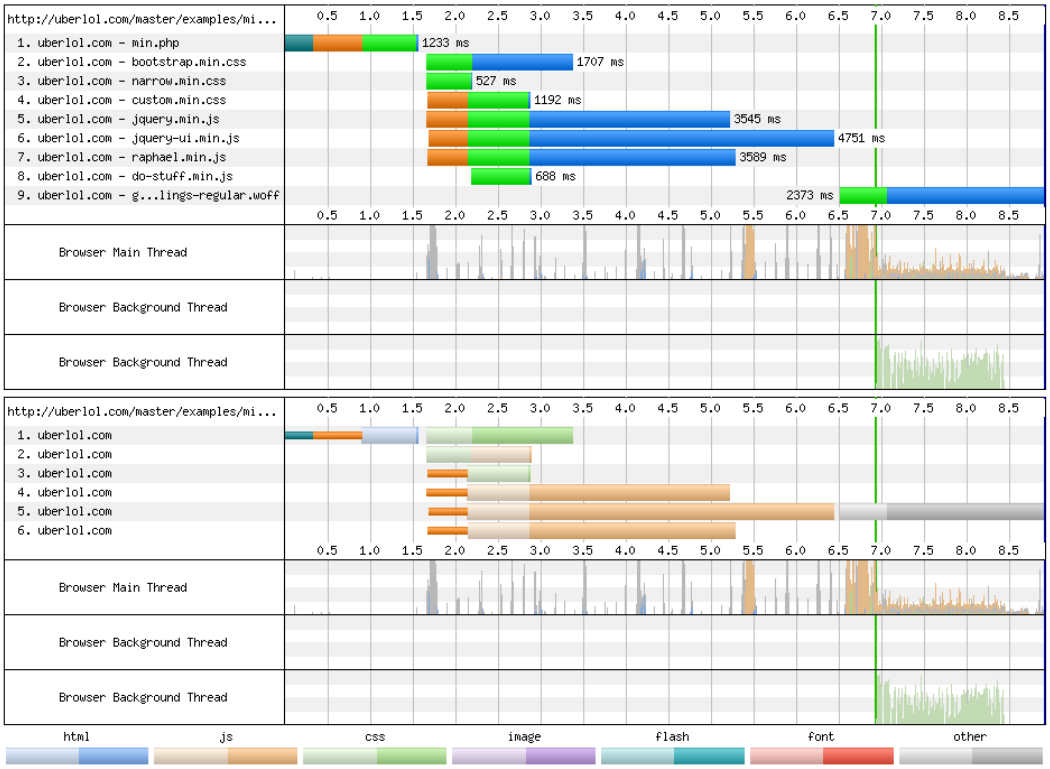


Figure 4.27: Mobile v2 – Post-test using minification

						Document Complete			Fully Loaded		
Load Time	First Byte	Start Render	Visually Complete	Speed Index	Result (error code)	Time	Requests	Bytes In	Time	Requests	Bytes In
9.470s	1.533s	7.842s	10.325s	8102	0	9.470s	4	299 KB	9.459s	4	299 KB

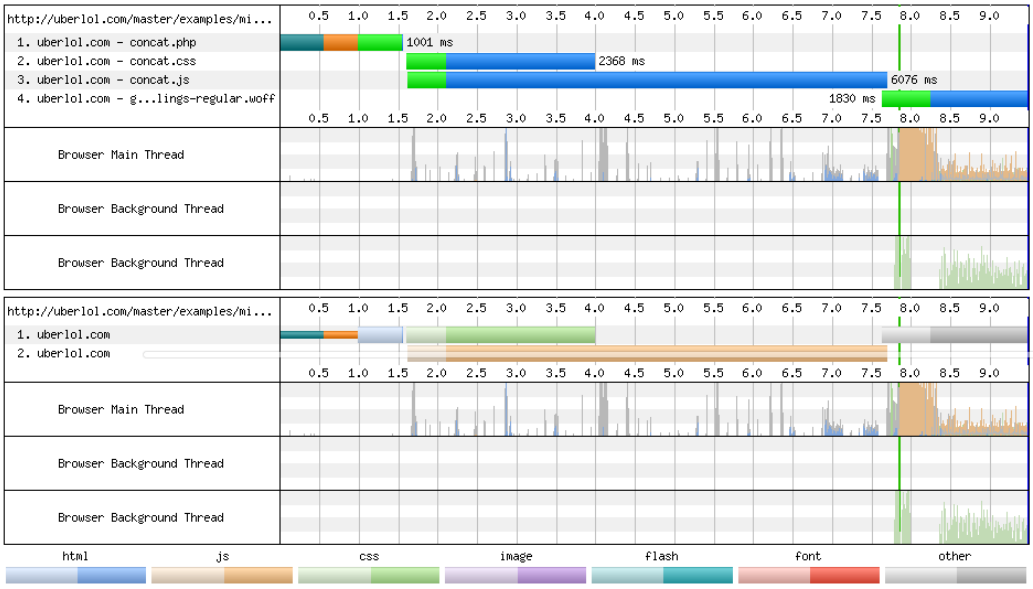


Figure 4.28: Mobile v2 – Post-test using concatenation

						Document Complete			Fully Loaded		
Load Time	First Byte	Start Render	Visually Complete	Speed Index	Result (error code)	Time	Requests	Bytes In	Time	Requests	Bytes In
6.528s	1.523s	6.332s	8.315s	6516	0	6.528s	4	164 KB	6.525s	4	164 KB

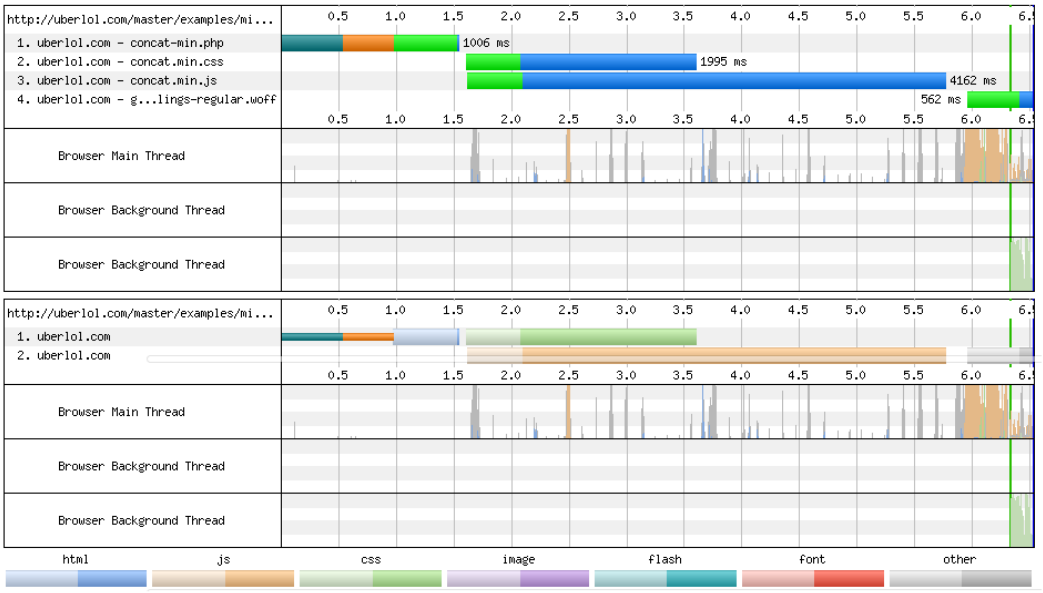


Figure 4.29: Mobile v2 – Post-test using both minification and concatenation

Results (desktop)

Technique	Start render (s)	Diff. pre-test (%)	Load time (s)	Diff. pre-test (%)	Total load size (KB)
Pre-test	1,995	–	4,906	–	384
Min	2,594	-30,03	3,180	35,18	249
Concat	3,592	-80,05	3,624	26,13	380
Min + concat	3,793	-90,13	3,925	20,00	246

Table 4.6: Desktop tests with compression

The pre-test using neither minification or concatenation (fig. 4.30) has a load time of 4,906 seconds, and a start render time of 1,995 seconds. Time to download all style sheets and scripts is 4,367 seconds.

Using minification (fig. 4.31), load time is improved by 1,726 seconds. Curiously enough, start render time is not improved, in fact it is 0,599 seconds slower. This can be attributable the `time to first byte`. Comparing fig. 4.30 and fig. 4.31 show that the time to download the main HTML (the first request) has a variation of 0,804 seconds. This variation can not be not be attributable to minification, since it does no affect the HTML. The extended time to first byte must therefore be caused by server or network conditions, which is outside of our control. If time to first byte is delayed, render and load times will also be delayed. This illustrates the danger of evaluating performance on render and load times alone, since they might be affected by external sources of error such as server response times and network conditions (signal routing and queuing delays). In these tests we therefore also look at the aggregate time to download all style sheets and scripts. In the case of minification and compression under desktop conditions (fig. 4.31), the time to download all style sheets and scripts is 1,012 seconds.

As mentioned in the mobile test, concatenation reduces the number of requests by 5, which saves time on server round trips as well as latency of establishing TCP-connections.

As a result we can see that bandwidth utilization is higher in fig. 4.32 than in the two previous tests. The graph indicates that data is transfered at a higher bit rate over a longer period of time than in the two previous tests. Still, concatenation has a smaller impact than minification. Load time is improved by 1,282 seconds, while start render time is 1,597 seconds slower than the pre-test. Slower render is again attributable to a delay in time to byte. The time to download all stylesheets and scripts is 2,093 seconds, which is 2,274 seconds faster than the pre-test (fig. 4.30) and 1,081 seconds slower than the post-test using minification (fig. 4.31).

The final test using both techniques (fig. 4.33) actually yielded slower load and render times then when minification and concatenation was applied individually. Load time is only improved by 0,954 seconds, while start render time is 1,798 seconds slower. The reasons for these results is not inherently clear. This test has approximately the same time to first byte the pre-test. In this case it seems like the delayed rendering time is due to the fact that the browser waits until all JavaScript is downloaded before starting to render the page. The pre-test starts rendering before all JavaScript is loaded. Oddly enough, the pre-test (fig. 4.30) is the only test which starts rendering before the DOMContentLoaded event is triggered (when all style sheets and scripts are downloaded, indicated by the pink line in the charts). This last test uses 3,217 seconds to download all all style sheets and scripts; quicker than the pre-test, but slower than the other post-tests.

							Document Complete			Fully Loaded		
Load Time	First Byte	Start Render	Visually Complete	Speed Index	DOM Elements	Result (error code)	Time	Requests	Bytes In	Time	Requests	Bytes In
4.906s	0.417s	1.995s	6.500s	3508	48	0	4.906s	9	384 KB	5.064s	10	384 KB

Figure 4.30: Desktop – Pre-test with no minification or concatenation

							Document Complete			Fully Loaded		
Load Time	First Byte	Start Render	Visually Complete	Speed Index	DOM Elements	Result (error code)	Time	Requests	Bytes In	Time	Requests	Bytes In
3.180s	1.311s	2.594s	4.000s	3069	48	0	3.180s	9	167 KB	3.331s	10	249 KB

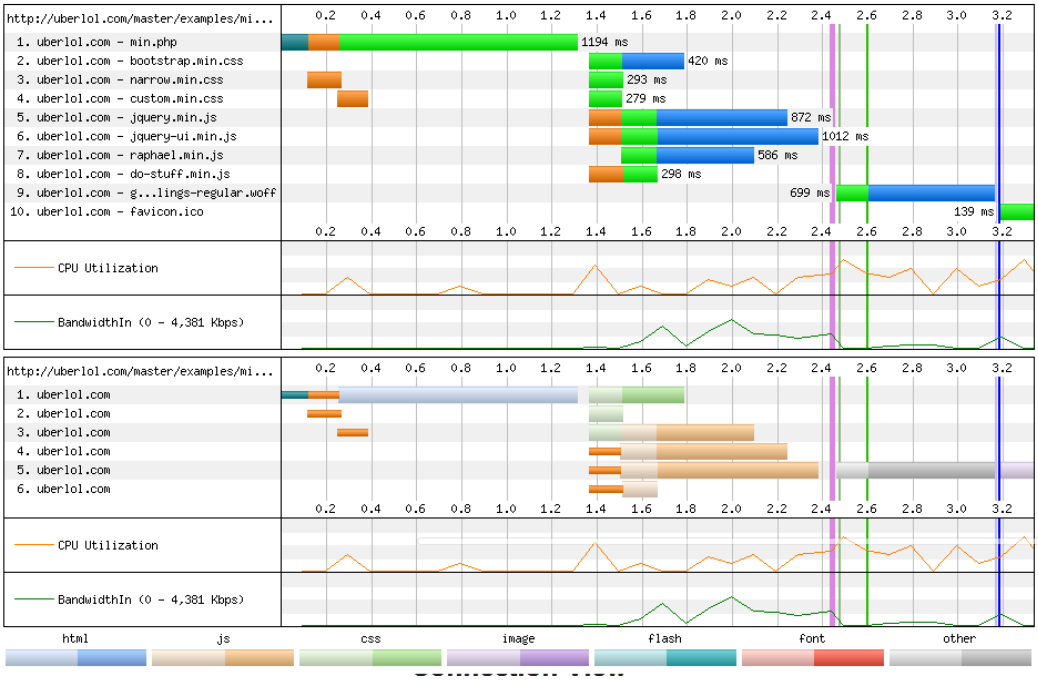


Figure 4.31: Desktop – Post-test using minification

Load Time	First Byte	Start Render	Visually Complete	Speed Index	DOM Elements	Result (error code)	Document Complete			Fully Loaded		
							Time	Requests	Bytes In	Time	Requests	Bytes In
3.624s	1.268s	3.592s	5.100s	4017	43	0	3.624s	4	380 KB	3.818s	5	381 KB

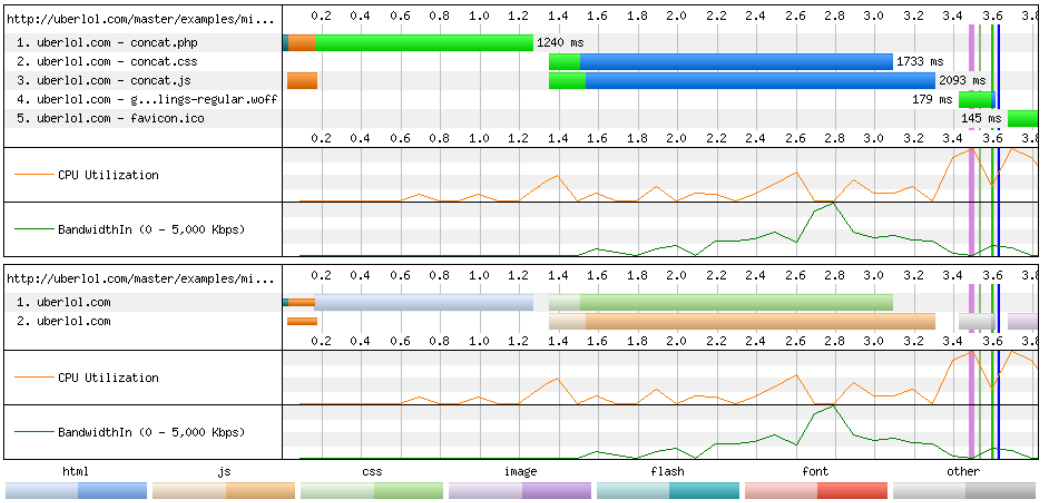


Figure 4.32: Desktop – Post-test using concatenation

Load Time	First Byte	Start Render	Visually Complete	Speed Index	DOM Elements	Result (error code)	Document Complete			Fully Loaded		
							Time	Requests	Bytes In	Time	Requests	Bytes In
3.952s	0.426s	3.793s	5.300s	4248	43	0	3.952s	4	246 KB	4.158s	5	246 KB

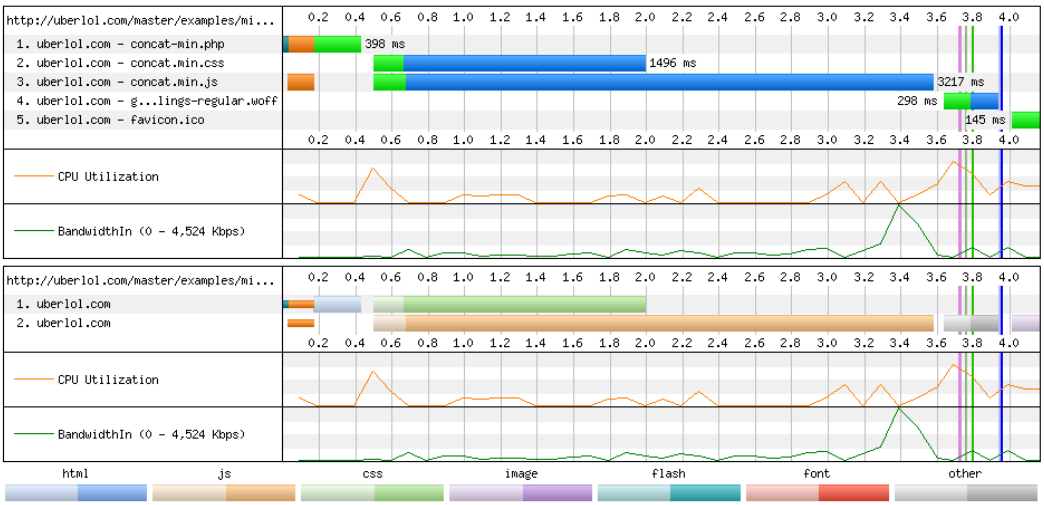


Figure 4.33: Desktop – Post-test using both minification and concatenation

4.4.2 Minification and concatenation without compression

The tests in this section looked at the effect of minification and concatenation while server-side compression was disabled. The procedure and sequence of tests are the same as in section 4.4.1. These test illustrate how big the impact of server-side compression is on file sizes, as well as how this affects overall performance.

Results (mobile v2)

Technique	**Start render (s)**	**Diff. pre-test (%)**	**Load time (s)**	**Diff. pre-test (%)**	**Total load size (KB)**
Pre-test	31,729	–	34,162	–	1220
Min	22,611	28,74	23,839	30,22	530
Concat	7,776	75,49	12,089	64,61	1211
Min + concat	8,104	74,46	9,651	71,75	529

Table 4.7: Mobile v2 tests without compression

The pre-test on mobile shows considerable differences when uncompressed. Overall load size is now more than four times the size then when compressed, 1220 KB (fig. 4.34) as compared to 302 KB (fig. 4.26). Load times of the mobile pre-tests differ by 19,670 seconds (uncompressed: 34,162s, compressed: 14,492s), while time to start render is 23,287 seconds slower (uncompressed: 31,729s, compressed: 8,442s). This highlights how big the impact of gzip compression is on file sizes and thereby load and render times.

Table 4.7 shows combined results for the four tests. When uncompressed the relative improvements on render and load times seem to shift compared to when compressed. Reading table 4.7 we see that the effect of minification is relatively small compared to that of concatenation.

The list below shows times to download all scripts and styles for each test:

1. Pre-test: 29,151 seconds (fig. 4.34)

2. Minification: 20,614 seconds (fig. 4.35)

3. Concatenation: 9,943 seconds (fig. 4.36)

4. Min. & concat: 7,519 seconds (fig. 4.37)

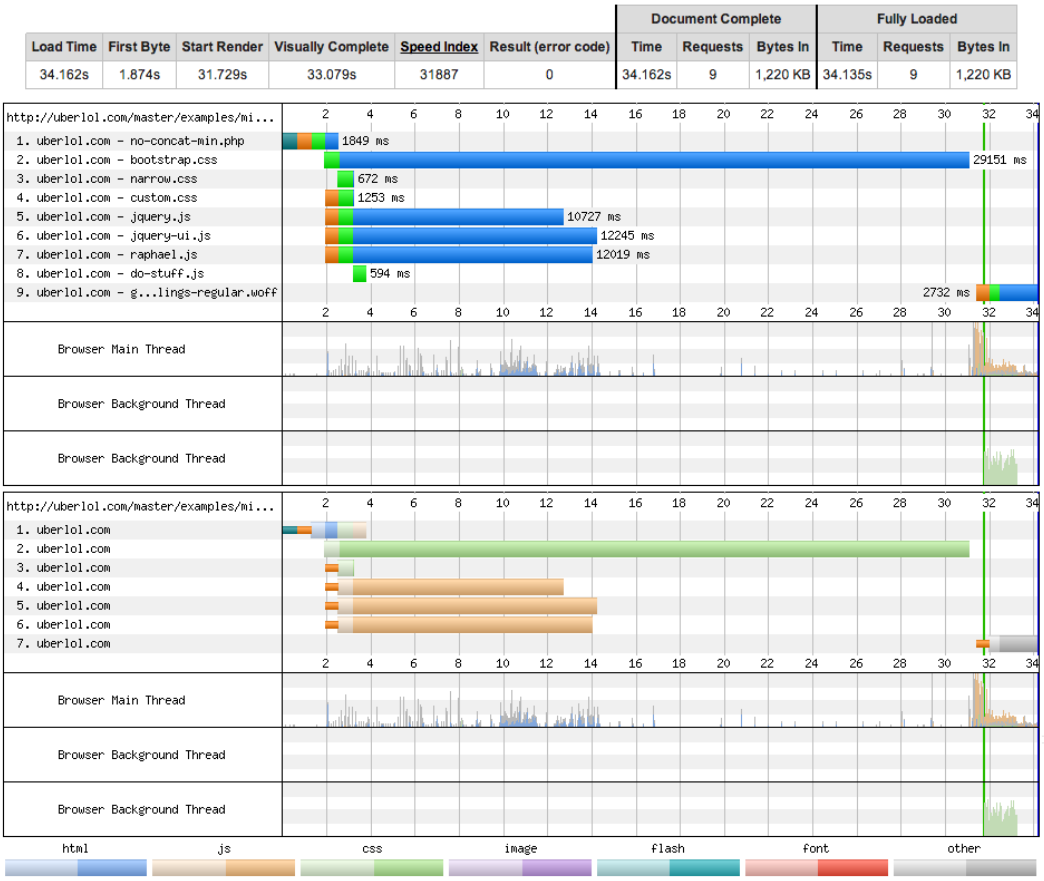


Figure 4.34: Mobile v2 – Pre-test with no minification or concatenation

						Document Complete			Fully Loaded		
Load Time	First Byte	Start Render	Visually Complete	Speed Index	Result (error code)	Time	Requests	Bytes In	Time	Requests	Bytes In
23.839s	1.281s	22.611s	24.061s	22757	0	23.839s	9	530 KB	23.829s	9	530 KB

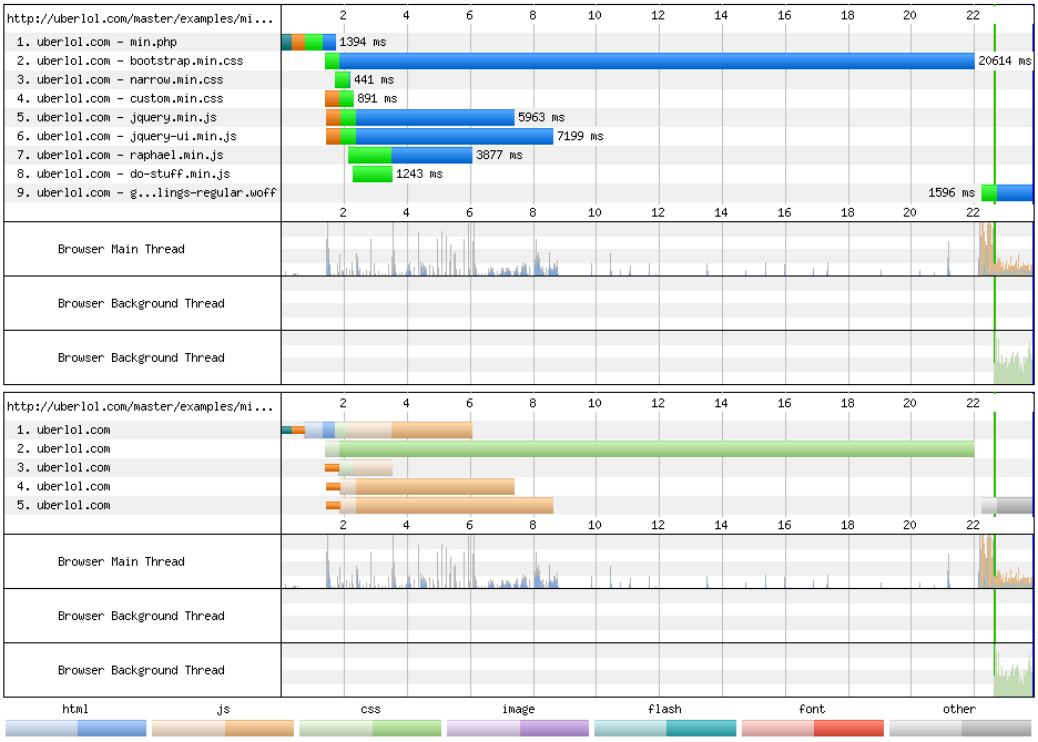


Figure 4.35: Mobile v2 – Post-test using minification

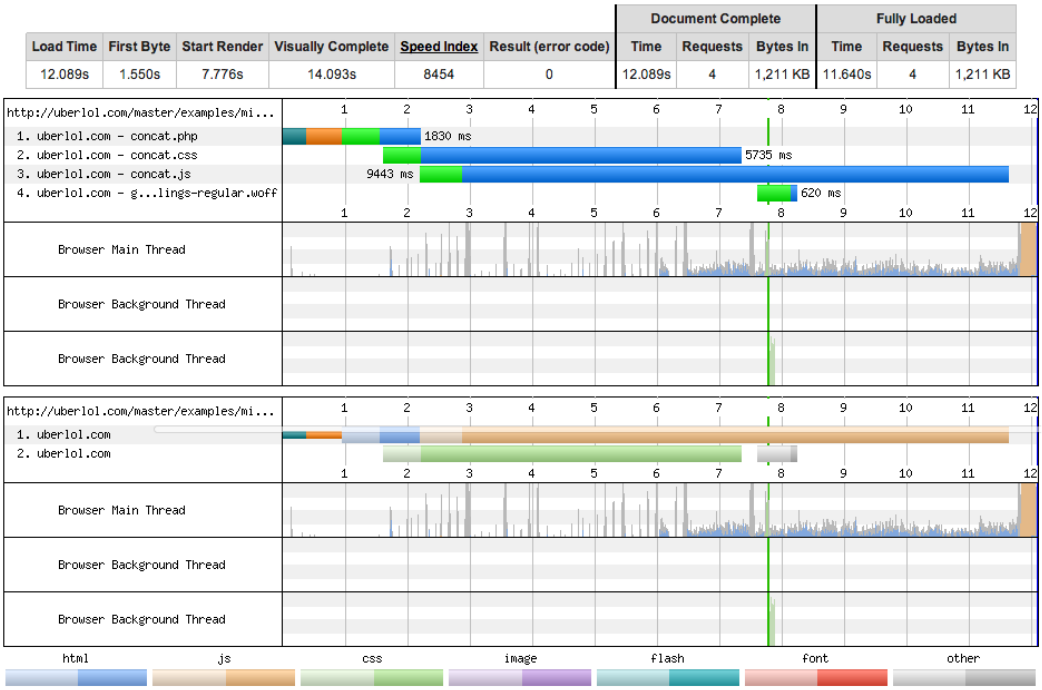


Figure 4.36: Mobile v2 – Post-test using concatenation

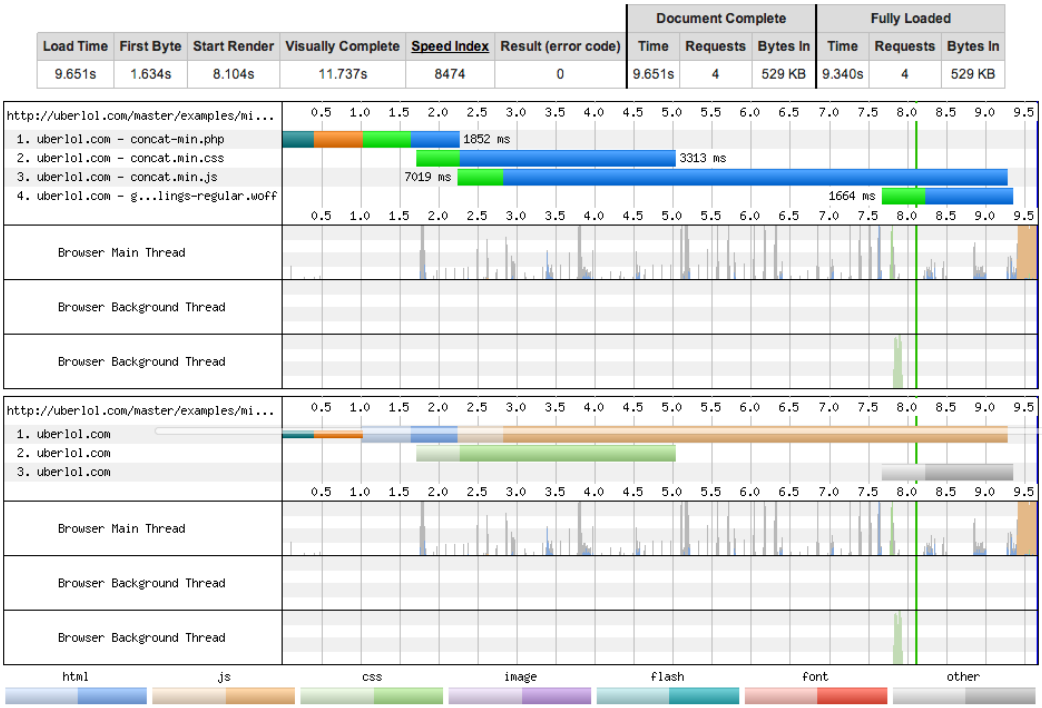


Figure 4.37: Mobile v2 – Post-test using both minification and concatenation

Results (desktop)

Technique	Start render (s)	Diff. pre-test (%)	Load time (s)	Diff. pre-test (%)	Total load size (KB)
Pre-tests	16,092	–	16,629	–	1302
Min	6,591	59,04	10,017	39,76	613
Concat	10,693	33,55	11,029	33,68	1294
Min + concat	4,493	72,08	4,589	72,40	612

Table 4.8: Desktop tests without compression

The uncompressed desktop tests affirm trends seen in previous tests. When comparing the pre-tests with (fig. 4.30) and without compression (fig. 4.38), we again observe that compression has a big impact on size, load and render times. Without compression load

size is 918KB larger, load time is 11,723 seconds slower, while start render is 14,097 seconds slower. These are similar results to what we saw when comparing mobile pre-tests.

Table 4.8 show that both minification and concatenation improves the results from the pre-test, but that in this case, minification has a bigger impact on performance. We again observe that combining both techniques yield better results, improving load and render times by more than 70% from the pre-test.

The list below shows times to download all scripts and styles for each test:

1. Pre-test: 14,857 seconds (fig. 4.38)
2. Minification: 8,940 seconds (fig. 4.39)
3. Concatenation: 9,943 seconds (fig. 4.40)
4. Min. & concat: 7,519 seconds (fig. 4.41)

Figure 4.38: Desktop – Pre-test with no minification or concatenation

							Document Complete			Fully Loaded		
Load Time	First Byte	Start Render	Visually Complete	Speed Index	DOM Elements	Result (error code)	Time	Requests	Bytes In	Time	Requests	Bytes In
10.017s	0.521s	6.591s	11.500s	8326	48	0	10.017s	9	612 KB	10.343s	10	613 KB

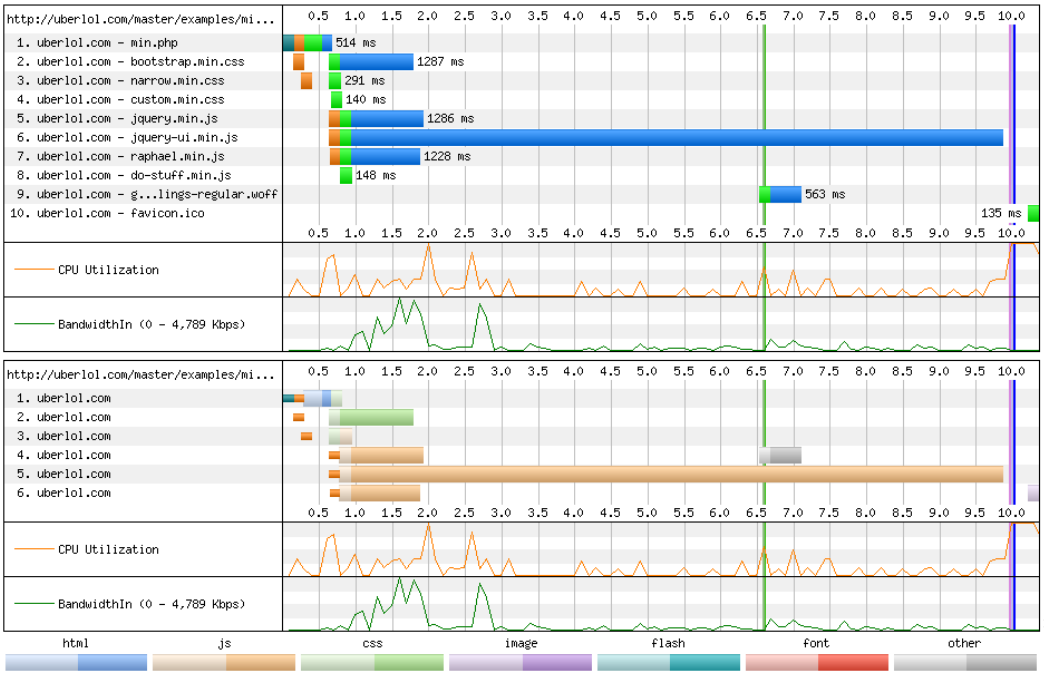


Figure 4.39: Desktop – Post-test using minification

Load Time	First Byte	Start Render	Visually Complete	Speed Index	DOM Elements	Result (error code)	Document Complete			Fully Loaded		
							Time	Requests	Bytes In	Time	Requests	Bytes In
11.029s	0.384s	10.693s	12.000s	11108	43	0	11.029s	4	1,293 KB	11.358s	5	1,294 KB

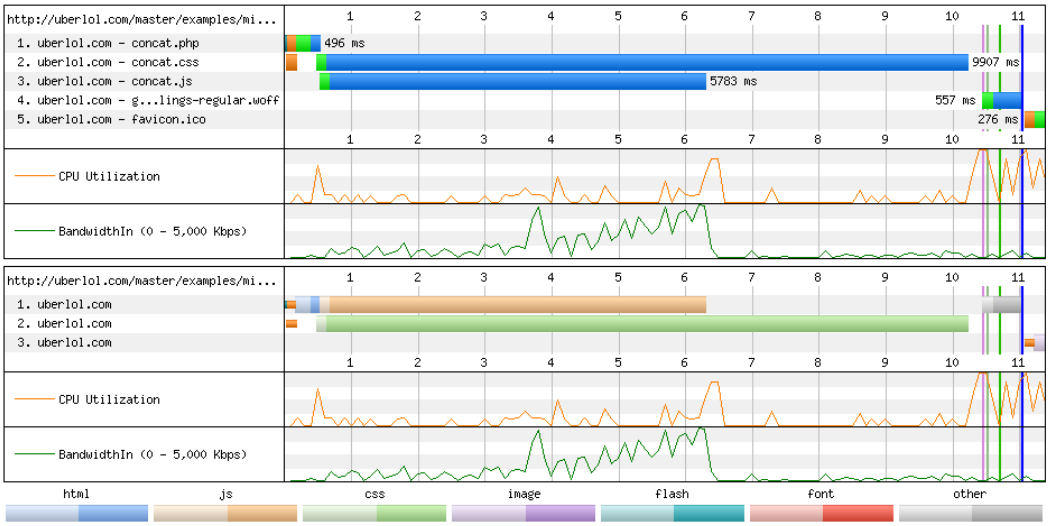


Figure 4.40: Desktop – Post-test using concatenation

Load Time	First Byte	Start Render	Visually Complete	Speed Index	DOM Elements	Result (error code)	Document Complete			Fully Loaded		
							Time	Requests	Bytes In	Time	Requests	Bytes In
4.589s	0.413s	4.493s	5.800s	4909	43	0	4.589s	4	611 KB	4.786s	5	612 KB

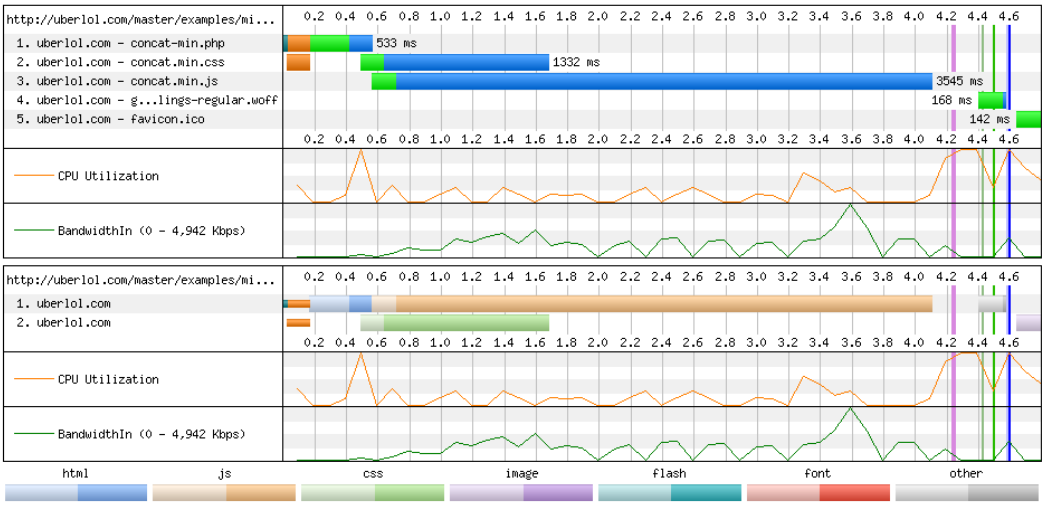


Figure 4.41: Desktop – Post-test using both minification and concatenation

4.4.3 Conclusion

The tests results in this section were somewhat scattered, and some tests results differed from expected behavior. A possible explanation is that results are susceptible to changing conditions and behaviors within the testing suite (WebPagetest). Because of irregularities in times to first byte, and because minification and concatenation was only applied to scripts and style sheets, the time to download only these resources was also used as a performance measurement. The collected times can be seen in fig. 4.42.

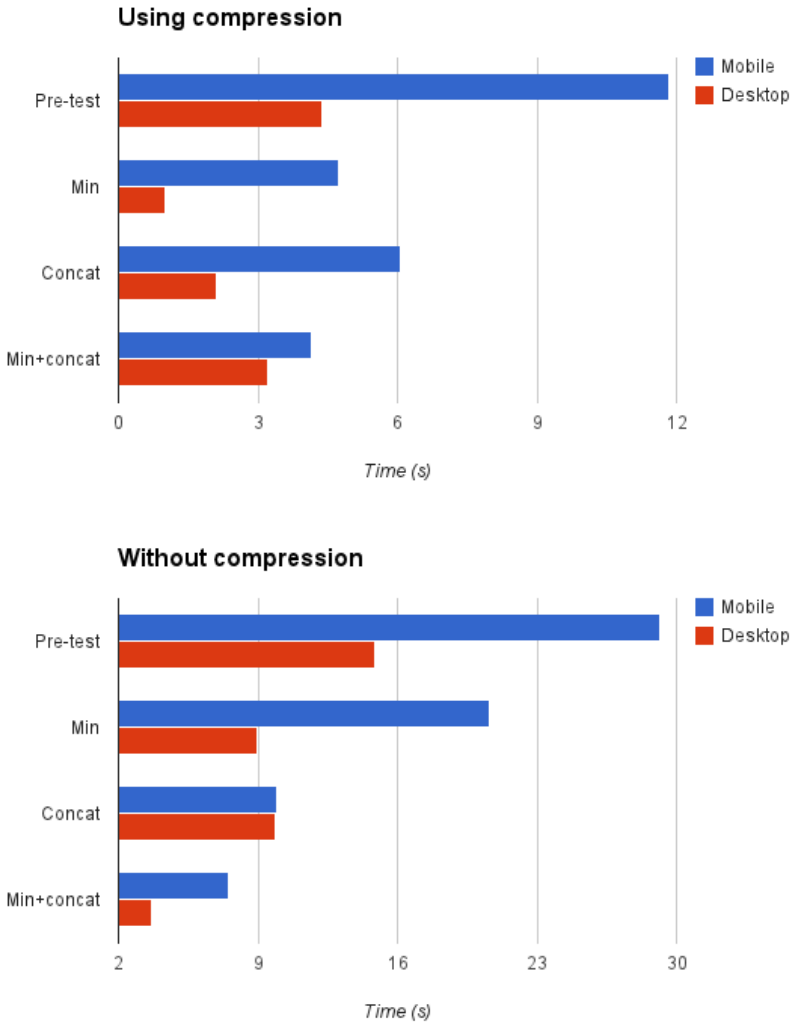


Figure 4.42: The time to download all stylesheets and scripts for the different tests

Since test results vary as much as they do, it is not possible to draw any definite conclusions. However, by comparing all the results, we are able to see some trends. These trends in large part correlate with the advantages highlighted in for the different techniques in chapter 3. The post-tests indicate that compression, minification and concatenation separately improve performance, some having a bigger impact than others. The tests also indicate that combining all three techniques yield better results than each technique by its own.

Compression can be regarded as a "low hanging fruit" with regards to front-end optimization. Because it is a server-automated process, it requires little effort to implement, in fact most servers have this feature activated by default. The impact of compression on overall performance is also seen to be quite large. In our tests, compression was seen to have the biggest overall effect on load sizes and times.

Compression alone reduced the combined file sizes of the CSS and JS files by 916,5KB (1194KB to 277,5KB), a 76,8% reduction. Combined reduction of both compression and minification was 1051,8KB (1194KB to 142,2KB), which is a 88,1% reduction. Minification used without compression reduced combined files sizes by 609KB (1194KB to 504 KB), which is a 57,7% reduction. This means that the relative reduction offered by minification is lower if the files are already compressed. This doesn't mean that minification isn't valuable, just that compression has a larger impact on file sizes. Both techniques should be employed for best overall results. Nicolaou (2013, p. 8) notes that under certain conditions, minification, when used in conjunction with compression, yield such minor improvements that it might not warrant its use. Still, our test results seem to indicate that minification offers a notable performance improvement even on top of compression.

Concatenating files reduced the number of overall requests by 5, which reduced the number parallel TCP connections to 2 instead of 6. This also had positive impacts on performance. However, the combined test results indicate that the performance improvements offered by compression and minification is bigger than that of concatenation. It

is surprising not to see bigger performance improvements when reducing the number of requests in the mobile context, where the round-trip times are high.

4.5 Improving server conditions

4.5.1 Domain sharding

To test the effects of domain sharding, an alias domain was created pointing to a directory on the host server. The test page included a series of images split across the main domain and the aliased domain. As a result the browser is "tricked" into increasing the number of concurrent TCP connections to the domain. The pre-test did not utilize the domain shard, and therefore loaded all resources from a single domain address (limiting the number of concurrent connections)

Results (mobile v2)

Fig. 4.44 shows that the browser opens 12 concurrent connections to download resources. It is interesting to note that the browser opens 8 connections in the pre-test (fig. 4.43). This indicates that this version of the Chrome mobile browser actually allows more than 6 concurrent connections per domain. The results show that the post-test using sharding (fig. 4.44) performs better with regards to both load and render times. Looking at the waterfall charts we can see that this is not directly a result of domain sharding, but rather because of differences in time to first byte between pre- and post-test. The load and render times are therefore most likely affected by network and/or server conditions. However, by only observing the time it takes to download images, we see that the post-test using sharding has a tendency to load the images slightly faster than the pre-test.

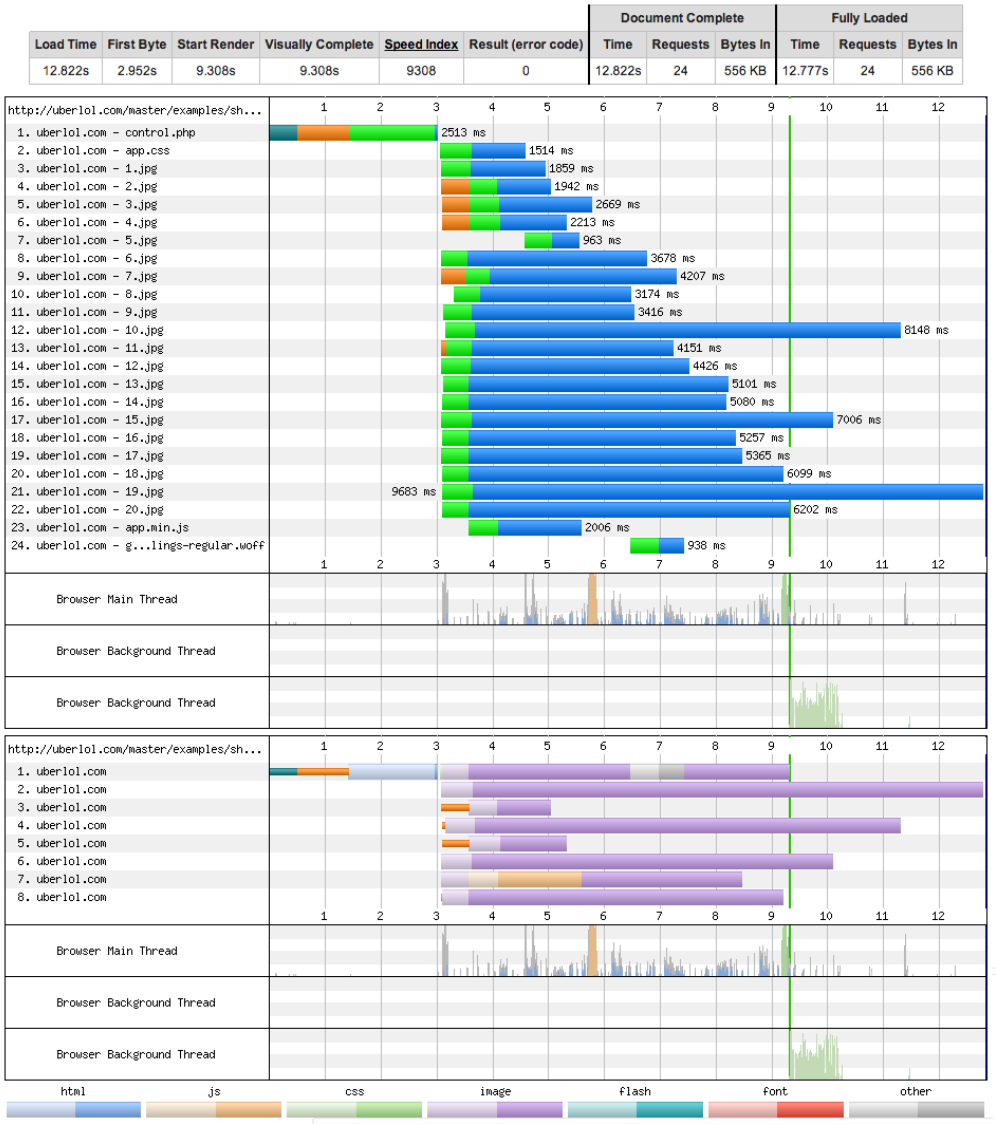


Figure 4.43: Mobile v2 – Pre-test loading all resources from single domain

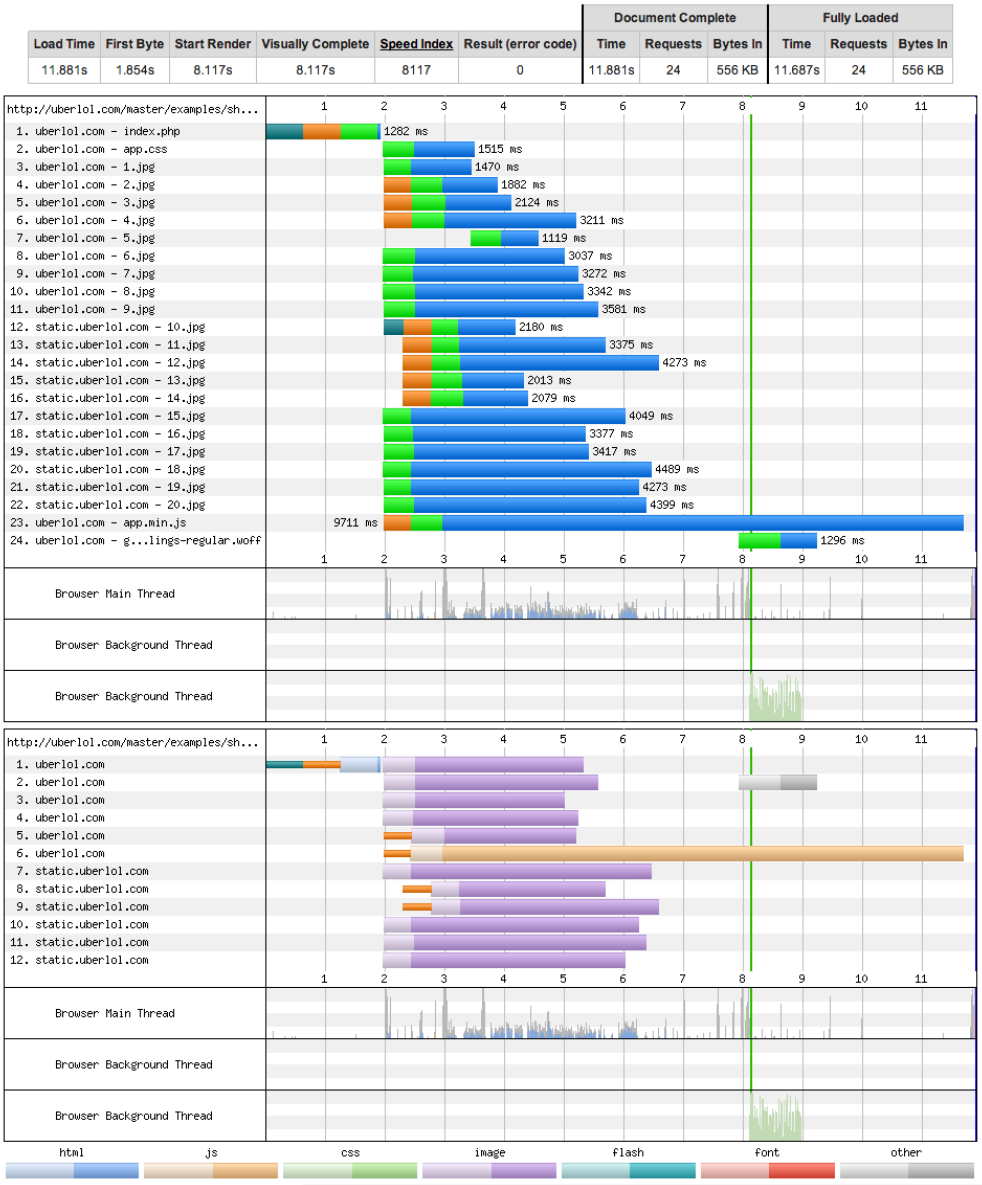


Figure 4.44: Mobile v2 – Post-test using domain sharding

Results (desktop)

Unlike in the mobile test, the desktop test more clearly indicate that sharding can directly affect performance. The pre- and post tests download the HTML document equally fast, indicating that server response and network conditions are similar. These tests show

improved results using sharding (fig. 4.46) than the pre-test (fig. 4.45) with regards to load and render time, and a shorter time to download all images.

Figure 4.45: Desktop – Pre-test loading all resources from single domain

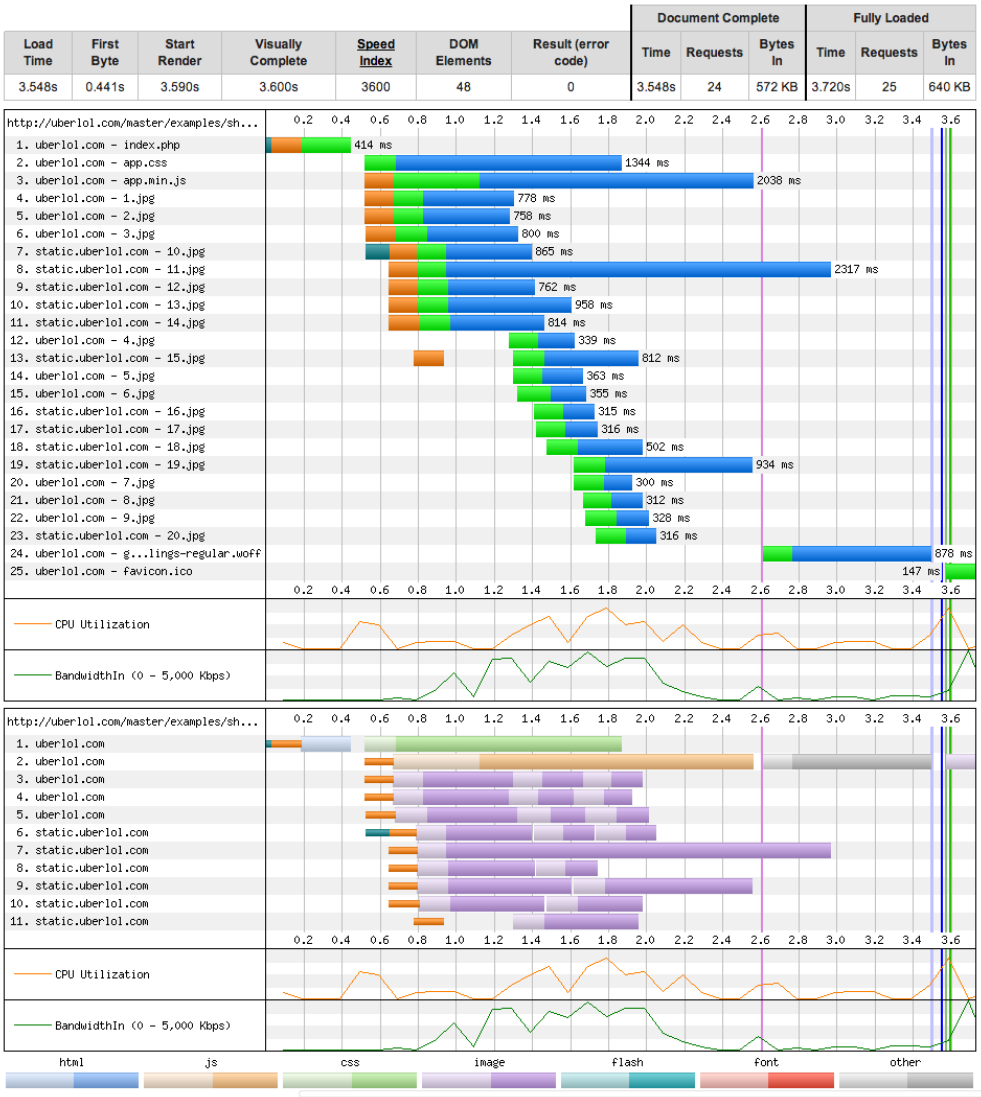


Figure 4.46: Desktop – Post-test using domain sharding

Conclusion

The test covered in this section indicate that sharding can have a positive effect on performance. These effects were more clearly seen in the desktop tests. Sharding most likely has lesser effect on mobile because of higher network delay (RTT). Establishing additional TCP connections therefore has a bigger penalty on performance.

4.5.2 Content delivery network

To test the effect of utilizing a CDN, we used the same setup and pre-test results as in the domain sharding test (section 4.5.1). Instead of serving static resources from our origin server we employed a commercial CDN provider[25] to host these resources. The CDN provider has a worldwide distribution network consisting of more than 34 data centers. Our origin server is located in Oslo, Norway, while our tests are run from Dulles, VA, in the US. By inspecting the response headers from these CDN tests we were able to see that our static files were served from a data center in Atlanta, GA. The air distance between Dulles and Atlanta (approx. 850 km) is considerably shorter than between Dulles and Oslo (approx. 6200 km). On the basis of proximity alone we therefore expected to see effects on download times.

The post-tests downloads all external resources specified in the main HTML from a CDN, not just images. This means that only the initial HTML response will be served from our origin server.

Results (mobile v2)

The post-test (fig. 4.47) shows better overall results on load and render times, compared to the pre-test (fig. 4.43). This can be partially attributed to the fact that the post-test has a shorter "time to first byte", and we see that the main document (.php) is downloaded 1,527 seconds faster in the CDN post-test. However, by looking at aggregate download times for all other resources, we see that downloading from the CDN is quicker. By comparing the pre-test (fig. 4.43) and post-test (fig. 4.47), we can observe that the latter resources downloaded in the post-test are downloaded noticeably faster than than in the pre-test (some several seconds faster).

[25] `http://www.cdn77.com/`

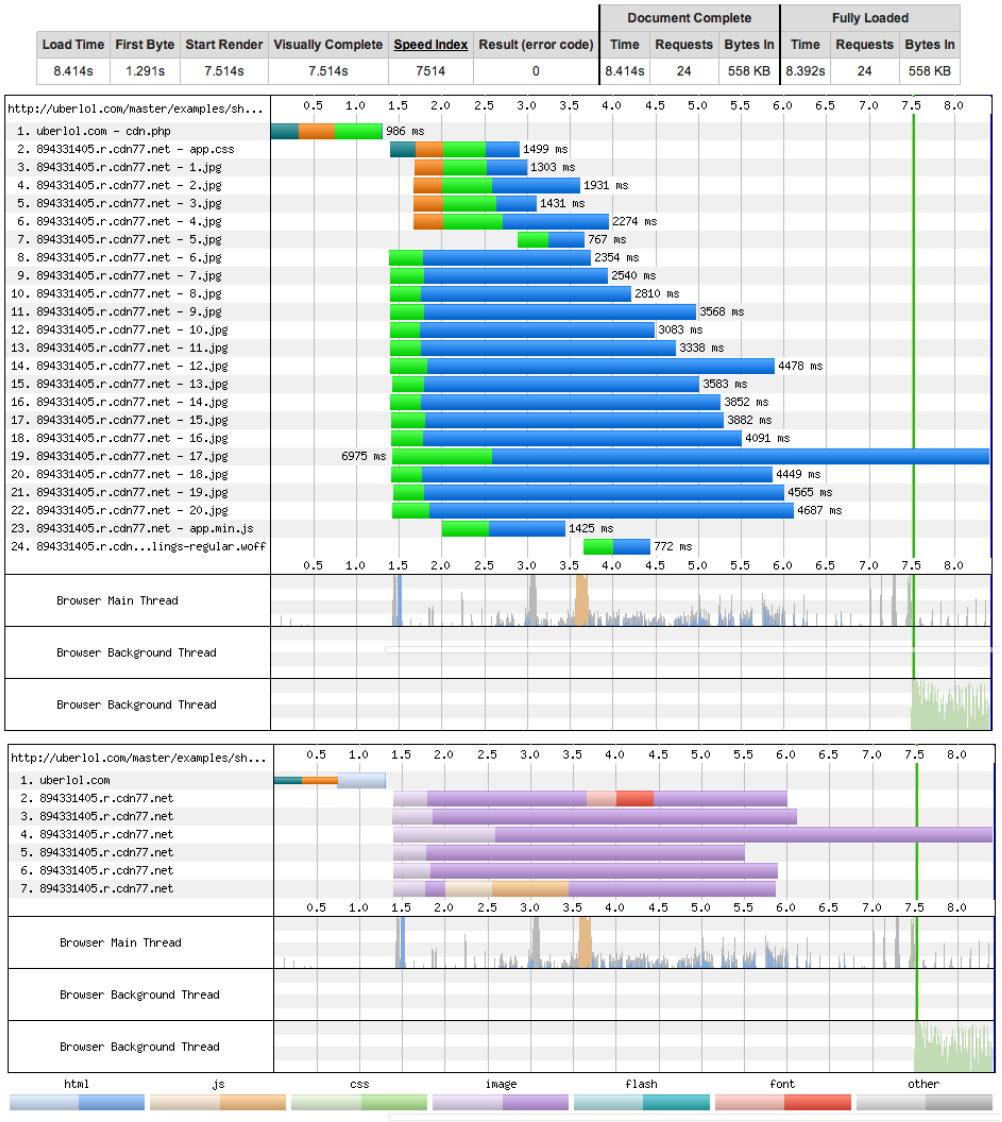


Figure 4.47: Mobile v2 – Post-test using CDN

Results (desktop)

The desktop post-test (fig. 4.48) shows similar results as the mobile post-test. The results here are easier to interpret since the HTML document is downloaded equally fast as in the pre-test (fig. 4.45). Time to first byte here only differs by 5 ms. We see that overall download times from CDN to client are shorter than from origin server to

client in the pre-test. In most cases the downloads are hundreds of milliseconds faster. From the graphs we can also see that the CDN test has a higher CPU and bandwidth utilization. This can indicate that resource delivery is more concentrated, and therefore more effective. Comparing results from the post-test we observe that start render time is 3,487 seconds faster, and load time is 3,444 seconds faster, than the pre-test

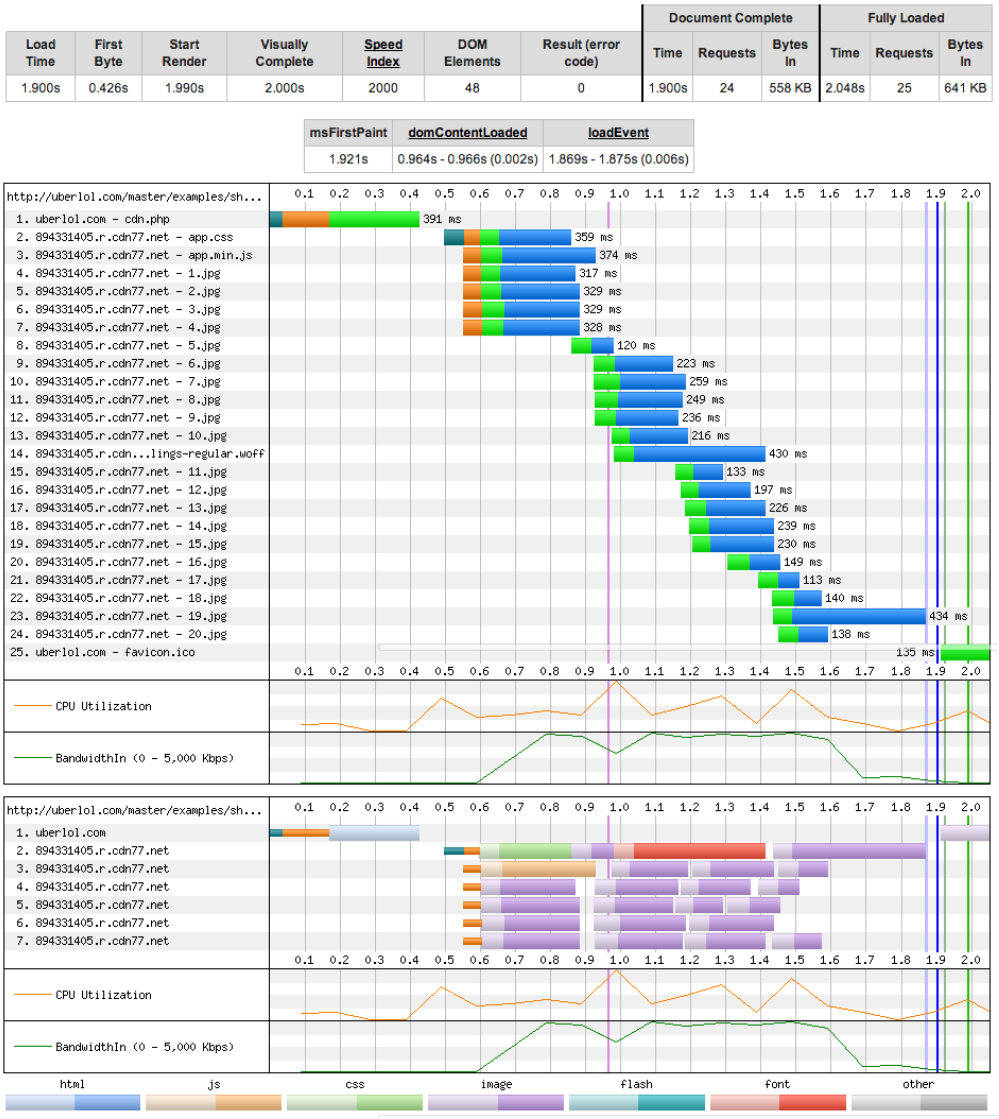


Figure 4.48: Desktop – Post-test using CDN

Conclusion

The results in these tests give clear indications that CDNs can improve resource load times, which again impacts overall performance. Use of a CDN had a positive impact under both mobile and desktop conditions. These results could be predicted from previous theoretical discussions on the effect of CDNs on network latency. Moving resources closer to the client geographically, will improve round trip times since the signal spends less time traveling between client and server. Shorter distances also means fewer intermediary connections where the signal is rerouted. This means that a network packet traveling a shorter distance has a lower probability of encountering performance bottlenecks along the way.

If the target demographics of a website is spread across the world, the use of a CDN will probably contribute to improved resource load times for most users. It would also help distribute server loads, since different servers will provide static resources for different geographic areas.

5 Discussion

5.1 Existing techniques

This thesis has examined several techniques that can be used to improve performance for responsive websites. We have looked specifically at how how to reduce requests, bytes, and server response times in order to improve web page load times. Each technique has been evaluated under mobile and desktop browsing conditions to examine their effect on web pages utilizing responsive design.

Our first research question asked *which techniques exist to improve front-end performance of device-agnostic websites*. While many different optimization techniques are available, this thesis has chosen to focus on techniques divided into four categories:

1. Implementing responsive images
2. Optimizing UI graphics
3. Optimizing textual resources
4. Improving server conditions

Implementing responsive images

The first category featured techniques that could be used to create responsive image solutions. A responsive image solution must be able to select appropriate image resources under different client conditions. These conditions include, but are not limited to, device screen size, browser viewport size, device orientation, and visibility of content. The techniques tested in this category were Picturefill, Adaptive Images, Clown Car technique, and lazy loading.

Picturefill is a JavaScript dependent polyfill mimicking the syntax of the proposed HTML5 `<picture>` element. Adaptive Images is a server-side solution written in PHP. It automates the task of substituting images under different conditions and also automatically

resizes and caches images to different dimensions. The Clown Car Technique is similar to Picturefill but utilizes SVG instead of JavaScript. The Clown Car technique allows a more modular approach to responsive images since it uses SVG breakpoints relative to parent layout elements, instead of device attributes like browser and viewport widths. The lazy loading technique by itself is not a responsive image solution, but allows web pages to only load images contained within the browser viewport.

Optimizing UI graphics

UI graphics are graphic elements considered part of presentation, not content. This includes both icons and background images. In this thesis two techniques for optimizing UI graphics have been considered: image sprites and symbol fonts. The aim of both solutions with regards to performance is to concatenate resources in order to reduce HTTP-requests.

Creating image sprites involves concatenating several raster images into a single file. Each individual image within the image sprite can then be mapped to coordinates using CSS. Symbol fonts allows the concatenation of several graphic elements into a single font file. The graphic elements are connected to different characters within the font and can therefore be referenced within both HTML and CSS.

Optimizing textual resources

Textual resources include, but are not limited to, HTML, style sheets and scripts. In order to optimize textual assets three techniques were considered: concatenation, minification and compression.

Concatenation involves combining several resources of the same type into a single file. Minification is a process of stripping code of all but the necessary characters. This involves removing whitespace, comments, and even refactoring code by shortening variable names and rewriting code to use shorthand properties (CSS). Compression is a server-side optimization technique where file sizes are reduced using compression techniques like

`gzip` and `deflate`. Files are compressed on the server before they are sent to a client.

Improving server conditions

Optimization of server conditions can be done by utilizing techniques to improve communication between client and server. In this thesis domain sharding and use of content delivery network have been evaluated to achieve this goal.

Domain sharding involves splitting web page resources over two or more domains. This is done to improve browser parallelization, i.e. increasing the amount of concurrent connections a client can have to the server(s). Content delivery networks (CDNs) are distributed networks of surrogate servers that duplicate content on the main (origin) server(s). The surrogate servers are distributed at many locations around the world, thereby bringing web page resources closer to clients. Closer proximity between client and server in turn ensures faster communication.

5.2 Considering advantages and disadvantages

Our second research question sought to elaborate on the functionality of the techniques identified by asking *which advantages and disadvantages must be considered when selecting and implementing these techniques.*

Implementing responsive images

Considerations that need to be taken when choosing a responsive image solution are:

- How flexible does the image selection logic need to be?
- Which use cases does the solution need to support?
- How easy is the solution to implement and maintain?
- Is it acceptable for images to be loaded later than other content?

Both Picturefill and the Clown Car technique (CCT) support usage of all CSS3 media features to adjust image selection logic. This includes, but is not limited to, `device-width`, `resolution`, `orientation`, and `width` (which in the case of Picturefill is the browser viewport width, and in CCT is the width of the parent element of the embedded SVG). Adaptive Images (AI) is limited to detecting device screen width and resolution when dictating image selection logic. Since Picturefill and CCT allows the user to explicitly specify which images to download under different conditions, they support the *art direction* use case. This means serving differently cropped images for different screen/viewport sizes. They also allow different images to be served for different mediums, such as screen or print. AI only target screens, and since image selection is automated only resized versions of different images can be offered.

The lack of customization offered by AI is compensated by its ease of use and implementation. Picturefill and CCT requires manual and redundant work specifying breakpoints and image paths; images also have to be resized by an external process. The AI script is able to do this automatically, saving time in the development process and maintenance of a website.

Another disadvantage to Picturefill and CCT is that they are dependent on loading, parsing and execution of other external resources before image downloads can start; JavaScript in the case of Picturefill, and SVG in the case of CCT. Since CCT is layout dependent it also has to wait on style calculations. These dependencies mean that images will start downloading later than other resources which affects total load times and perceived performance (images become visible after other content). Since images start loading later than other resources, browser parallelization is reduced, i.e. the browser utilizes fewer concurrent TCP-connections.

Lazy loading introduces a deliberate delay in downloading images not present within the browser viewport. This can greatly reduce load size on initial page load. However, since space is not reserved for unloaded images, this technique is challenged by disruptive layout reflows when images are loaded after initial load. To counter this effect lazy loading

should be combined with an image placeholder technique like the "padding-bottom hack". Implementing an image placeholder technique adds complexity and can potentially also affect performance.

Optimizing UI graphics

Both image sprites and symbol fonts allow several graphic resources to be concatenated into a single file. This reduces the amount of HTTP-requests needed on a web page. As a result, the tests performed showed improvement in load times for both techniques.

Symbol fonts have the advantage of possessing the same characteristics as text based fonts. This means that they are vector based and therefore infinitely scalable. They are also easily manipulated through CSS, giving them great flexibility in appearance. Image sprites are raster based (bitmap) and are therefore not as easily scaled or manipulated. Since image sprites need to coordinate mapping in CSS they are also harder to resize and and edit. A limitation of fonts is that they are monochromatic, and therefore don't work well for multicolored graphics. Graphics like photographs are also hard to replicate in vector format because of their random nature. Photographic background images therefore lend themselves better to a raster format like image sprites.

Optimizing textual resources

Concatenation allows multiple resources to be combined to a single file thereby reducing HTTP-requests. Minification and compression reduce file sizes, allowing resources to be transferred in fewer network packets.

Our test showed that compression and minification has a large impact on file sizes. Use of `gzip` compression alone reduced the combined size of scripts and style sheets by 76,8%. Using minification alone gave a reduction 57,7%, while combining compression and minification resulted in total reduction of 88,1%. Compression and minification use different techniques to reduce file sizes, and therefore yield the best results when combined. Compression can be applied to HTML documents as well, while minifying

HTML is harder because it is dependent on whitespace and conditional comments[26]. Using compression and minification can yield considerable performance gains, and have few disadvantages.

Testing showed that concatenation reduced the number of HTTP requests, resulting in slightly faster load times. However, it was surprising to see that concatenation did not have a bigger impact on load times under mobile conditions. Concatenated scripts and style sheets also take longer time to parse. Because of this precautions need to be taken when concatenating files in order to prevent negative impacts on render time. Concatenation can also be a challenge with regards to caching since it prevents individual caching of resources.

Improving server conditions

Splitting resources over several domains allows the browser to open more concurrent TCP connections. In this thesis, domain sharding was implemented by serving half of a web page's resources from a domain subdirectory. As a result the browser utilized more concurrent connections, which showed an improvement in resource load times. Domain sharding needs to be done in moderation since each new shard introduces performance overhead with regards to DNS-lookups and establishment of new TCP-connections. Other research therefore recommends that aggressive sharding should be avoided, and that no more than two domain shards should be used (McLachlan, 2012; Souders, 2013). Basic domain sharding (on a single server) is easy to implement and does not involve any additional development costs.

Using a CDN allows resources to be downloaded from a server with closer proximity to the client. In testing this resulted in faster load times for the resources loaded from CDN (surrogate) servers, than when the same resources were loaded from the origin server. Using a CDN also allows server loads to be split across multiple surrogate servers,

[26]Conditional comments are special styling instructions specifically targeting Internet Explorer prior to version 10.

distributing server workload.

CDN disadvantages include cost, and complexity of resource distribution. Using a CDN usually involves paying a CDN provider for storage and network traffic. The positive effects of CDN usage therefore needs to be evaluated against these costs. If a website targets a large global demographic the performance gains afforded by a CDN probably easily justify costs. However, if a website only targets a national demographic other approaches might be more suitable since proximity to clients might not be improved by a CDN. Another aspect to consider is the fact that CDN content distribution is handled by an external provider. This means that the CDN provider controls how the surrogate servers cache content from the origin server.

5.3 Limitations of research methodology

This section will talk briefly about how the results of testing might have been affected by limitations in research methodology.

The quality of the experiments research conducted in this thesis can be measured under the following criteria (Oates, 2005, p. 287):

- **Objectivity**

  The ability to avoid influencing the results, and to ensure that the research is free of bias and distortions.

- **Reliability**

  Ensuring that instruments and measurement are neutral, accurate and reliable. Repeatability should ensure that the results are consistent and unambiguous, and that experiments could be replicated and yield the same results.

- **Internal validity**

  Ensuring that the research was well designed and that the right data was collected from the right sources. Ensuring that the causal links found through experimentation are valid.

- **External validity**

  Ensuring that the findings and conclusions are generalizable, meaning that they can be used by other people under different circumstances and still be valid.

When evaluating different performance techniques both advantages and disadvantages have been considered and discussed. Experiments (performance tests) have been performed in order to supplement the discussions in order to identify correlations between the two. The combination of theoretical discussion and experimental testing has sought to give an *objective* evaluation of each performance optimization technique.

Each technique covered has undergone 2 rounds of pre- and post-testing, one using a mobile browser on a mobile network connection, the other using a desktop browser on a cabled network connection. The pre-test (or control test) looked at the performance of the system before a technique was applied, while the post-test looks at performance after the application of a technique. The purpose pre- and post-testing was done to identify changes (effects) linked to causing factors (Oates, 2005, p. 131). By running the tests multiple times under different conditions, we aimed to improve the *repeatability* of our results. We have tried to make the test process as transparent as possible by describing the system and test conditions, using freely available technologies, and by giving a detailed account of results through discussion and visual aides. By making the process more transparent we aim to improve the reproducibility of our results.

The markup used for the web pages in the performance tests was kept fairly simplistic, in hopes of isolating the effect of each performance optimization technique by itself. This approach is similar to reductionism within the scientific method, a technique were

complex systems are broken down to smaller entities which are more easily studied (Oates, 2005, p. 285). However, as the test results seemed prone to the effect of unknown or uncontrollable factors (network delay, server response times, errors in code etc.) the results can not be said to be entirely reliable. In some cases results where slightly contradictory to theory. We therefore analyzed the tests results and tried to explain unexpected behavior. Since we were not able to control all aspects of the external test system it is hard to identify all potential sources of error. Because of these conditions, it was harder to ensure the *internal validity* of all results.

The thesis would therefore have benefited from more rigorous test conditions. The tests conducted were all performed using a single system (WebPagetest), which also constitutes a single point of failure. If not for time restrictions, the results of the tests performed should have been verified against a second test system, preferably one were different variables were more easily controllable. One example of such a system could have been a server instance (e.g. LAMP) running locally on a machine utilizing bandwidth simulation software. The result of tests on browser instances in a simulated network could then have been compared with results observed on real networks and devices (offered by WebPagetest). More time could also have been spent identifying the margin of error on the different test instances offered by the system.

The fact that each test consisted of a simplified usage scenario might have impacted the generalizability, and thereby the *external validity* of the results. The danger of using inspection experiments that asses easily quantifiable benefits in a simplified scenario is that implications on performance of a more complex system cannot be assessed because "optimization in one dimension can suboptimize overall performance" (Kitchenham et al., 2002, pp. 723-724). It is therefore not possible to state that optimization effects witnessed in our static web page experiments will give exactly same results on dynamic websites with more complex structures and compositions of external resources. Still, the thesis contributes a set of results that could be used as a basis for further evaluation and research.

Other considerations

In testing, total load times were often used a main performance metric, but as witnessed through testing this was not always reliable. Certain uncontrollable variables, like server response times and network latency, can result in longer times to initiate connections (time to first byte), which prolongs the total load time. It is also important to note that total load times does not necessarily reflect user experience. This is because in many cases, rendering and interaction with a web page can commence long before all resources have been downloaded. The thesis therefore observes some metrics for performance, but does not delve too much into how to improve *perceived* performance, which can be even more important from a user perspective.

One big aspect of web performance that has not been evaluated in this thesis is browser caching. Caching probably has the single biggest impact on performance for repeating visitors since it removes the need for resources to be downloaded from the network. Many different techniques exist to improve the cacheability of resources, but in order to narrow down discussion and testing, it was chosen to focus on optimization techniques for first-time visitors with empty browser caches. Research performed by Yahoo in 2007 (Theurer, 2007) also showed that as much as 60% of their visitors requested pages on empty browser caches. This stresses the importance of designing for the empty cache user experience.

5.4 Future developments

The HTTP 2.0 specification is planned for release as a proposed standard in November 2014 (IETF, 2014). This update of the protocol will solve many previous shortcomings of HTTP 1.1. It will no longer be necessary to concatenate files because of the introduction of a new feature known as *request and response multiplexing*. Multiplexing allows for all data transferred between client and server to go through a single connection, thereby eliminating much unnecessary latency (Grigorik, 2013a, pp. 45-46). With request and response multiplexing, domain sharding would also be rendered useless, and would ac-

tually harm performance as it incurs additional DNS-lookups. Compression is also done by default over HTTP 2.0.

Another feature of HTTP 2.0 that will undoubtedly improve performance is the concept of *server push*. Where the browser would previously have to send GET-requests to download each asset in an HTML document, the server will now sends those assets automatically along with the HTML document (Grigorik, 2013a, p. 47). This means that a GET-request for an HTML document would retrieve not only the HTML, but several other assets like CSS, images, JS, etc. This will greatly minimize the necessity of server round trips, and diminish the effect of network delays.

Certain optimization techniques would still be needed regardless of HTTP 2.0. These include usage CDNs (that improve response times and RTTs), as well as responsive images and any form of selective loading which can reduce load sizes appropriately for different devices.

Responsive image solutions are also under current development, and reports indicate that both the `<picture>` element and `srcset` attribute are close to seeing actual browser implementation (Weiss et al., 2014). In fact a subset of the `srcset` attribute (allowing resolution based image switching) has been implemented in experimental versions of both WebKit (Jackson, 2013) and Chrome (Toy, 2014) desktop browsers. These browser-native solutions will ensure that image selection will happen during HTML parsing, and will therefore allow images to download simultaneously with other resources. This is unlike current script-dependent techniques which have to wait for script download, parsing and execution before image downloading can start.

Even though the implementation of these technologies will undoubtedly improve web performance, it will still take time before any of them have gathered wide support on web clients and servers. Using the techniques discussed in this thesis will therefore still be a valid option until the technologies have reached wide adoption. HTTP 2.0 might see implementations in early 2015, and since the update from version 1.1 to 2.0 said to be easily applicable on current web architecture it will probably see a fast adoption.

Because of this, HTTP 1.1-specific workarounds like concatenation and domain sharding should perhaps be abandoned in the near future.

6 Conclusion

6.1 Summary

This thesis has sought to identify performance optimization techniques for websites, and to evaluate their inherent advantages and disadvantages. Particular effort has been put into identifying front-end performance optimization techniques that are particularly beneficial to responsive websites under mobile browsing conditions. The techniques highlighted in the thesis can be divided into four categories: implementation of responsive images, optimizing UI graphics, optimizing textual resources, and optimizing server/client communication. The aim of these techniques is to improve performance by reducing bytes downloaded, reducing HTTP-requests and by accelerating communication between client and server.

Testing showed that it is possible to implement a responsive image solution using existing technology, and that all these solutions can be beneficial to performance under mobile browsing conditions. However, no technique evaluated was without distinct disadvantages, and usually had a slight negative impact on performance under desktop conditions.

More established optimization techniques were also tested to see the effect of file concatenation, file size reduction, increasing concurrent connections, and reducing proximity between client and server. The results show that reducing file sizes through compression and minification has high gains and very few risks. Concatenation is shown to reduce HTTP-requests, but at the cost of reducing cacheability. Increasing concurrent connections through domain sharding can improve performance in some scenarios, but is advised to be used in moderation because of the possibility for negative side effects. Using content delivery networks allow for faster communication between client and server under certain conditions. It also distributes server load, but comes at a financial cost.

6.2 Further research

This thesis has used performance metrics and waterfall charts collected from an online testing as one way of evaluating performance. However, computer measured data does not necessarily reflect user perceived performance. It would therefore be interesting to research how performance optimization techniques was perceived by actual users in real-world scenarios. User testing could for instance be performed asking users to complete different tasks under varying conditions to evaluate if and how increased web performance aided quicker task completion. Interviews could also be performed to identify which performance optimization techniques that have the biggest impact on user experience.

Further research could also look deeper into RUM (real user measurement) (Meenan, 2013), where scripting is used to gather detailed performance statistics from real user interaction with a website. Doing this kind of research could include A/B testing of different optimization techniques on a highly trafficked website in order to obtain data. The data could then be evaluated through quantitative data analysis.

The proliferation of JavaScript driven web applications could also be an interesting foundation for web performance research. This thesis only looks at static websites utilizing minimal JavaScript, while some web applications might be driven solely by JavaScript. It would therefore be interesting to evaluate optimization techniques for DOM manipulation, script loading order and dependencies, and use of synchronous and asynchronous JavaScript.

Several new web technologies are scheduled to be released within the coming year. When released, it would be interesting to do preliminary research on the effect of the HTTP 2.0 protocol, as well responsive image solutions like the `<picture>` element, the `srcset` attribute, and content negotiation through HTTP Client Hints. Grigorik (2013a, p. 49) cites "optimal implementations of header-compression strategies, prioritization, and flow-control logic both on the client and server, as well as the use of server push" as potential research domains for HTTP 2.0.

References

A. Andersen and T. Järlund. Addressing the responsive images performance problem: A case study. *Smashing Magazine*, September 16, 2013. URL http://www.smashingmagazine.com/2013/09/16/responsive-images-performance-problem-case-study/.

Apache. Apache http server tutorial: .htaccess files, 2014. URL http://httpd.apache.org/docs/2.2/howto/htaccess.html.

J. Archibald. Response times: The 3 important limits. *HTML5 Rocks*, June 5, 2013. URL http://www.html5rocks.com/en/tutorials/speed/script-loading/.

K. Azad. How to optimize your site with gzip compression, April 4, 2007. URL http://betterexplained.com/articles/how-to-optimize-your-site-with-gzip-compression/.

S. Basu. Source maps 101. *Tutsplus*, January 16, 2013. URL http://code.tutsplus.com/tutorials/source-maps-101--net-29173.

R. Berjon, S. Faulkner, T. Leithead, E. D. Navara, E. O'Connor, S. Pfeiffer, and I. Hickson. Html5 - a vocabulary and associated APIs for HTML and XHTML. W3C candidate recommendation, W3C, April 29, 2014. URL http://www.w3.org/TR/2014/CR-html5-20140429/.

B. Bos, T. Çelik, I. Hickson, and H. k. W. Lie. Cascading Style Sheets Level 2 Revision 1 (CSS 2.1) Specification, 2009. URL http://www.w3.org/TR/CSS2/.

M. Caceres. Paris responsive images meetup, September 18, 2013. URL http://www.w3.org/community/respimg/2013/09/18/paris-responsive-images-meetup/.

Can I Use. Can i use svg, 2014. URL http://caniuse.com/#cats=SVG.

S. Champeon and N. Finck. Inclusive web design for the future, 2003. URL http://www.hesketh.com/thought-leadership/our-publications/inclusive-web-design-future.

W. Chan. Network congestion and web browsing, May 20, 2013. URL https://insouciant.org/tech/network-congestion-and-web-browsing/.

Cisco. Visual networking – global mobile data traffic forecast update, 2012-2017, 2013. URL http://www.cisco.com/en/US/solutions/collateral/ns341/ns525/ns537/ns705/ns827/white_paper_c11-520862.html.

Coding Capital. Does firefox support mp4? URL http://www.codingcapital.com/webpreneur/entrepreneurship/does-firefox-support-mp4/.

A. Davies. *Pocket Guide to Web Performance*. Five Simple Steps, Penarth, UK, 2013a.

A. Davies. How the browser pre-loader makes pages load faster, September 2, 2013b. URL http://andydavies.me/blog/2013/10/22/how-the-browser-pre-loader-makes-pages-load-faster/.

I. Devlin. Responsive HTML5 video, August 20, 2012. URL http://www.iandevlin.com/blog/2012/08/html5/responsive-html5-video.

Y. Elkhatib, G. Tyson, and M. Welzl. The effect of network and infrastructural variables on spdy's performance. *arXiv preprint arXiv:1401.6508*, 2014.

K. Fehrenbacher. M dot: Web's answer to mobile. *Gigaom*, May 11, 2007. URL http://gigaom.com/2007/05/11/m-dot-webs-answer-to-mobile/.

J.-l. Gailly and M. Adler. What is gzip?, 1999. URL http://www.ee.uwa.edu.au/~roberto/teach/itc314/resources/gzip.txt.

T. Garsiel and P. Irish. How browsers work: Behind the scenes of modern web browsers. *HTML5 Rocks*, August 5, 2011. URL http://www.html5rocks.com/en/tutorials/internals/howbrowserswork/.

Google. Minimize round-trip times, 2014a. URL https://developers.google.com/speed/docs/best-practices/rtt#MinimizeDNSLookups.

Google. Dom construction process [image], 2014b. URL https://developers.google.com/web/fundamentals/documentation/performance/critical-rendering-path/images/full-process.png.

I. Grigorik. Making the web faster with http 2.0. *Commun. ACM*, 56(12):42–49, Dec. 2013a. ISSN 0001-0782. doi: 10.1145/2534706.2534721. URL http://doi.acm.org/10.1145/2534706.2534721.

I. Grigorik. *High Performance Browser Networking: What Every Web Developer Should Know about Networking and Web Performance*. O'Reilly Media, Inc., 2013b.

A. Gustafson. Progressive enhancement with css. *A List Apart*, October 2008a. URL http://alistapart.com/article/progressiveenhancementwithcss.

A. Gustafson. Understanding progressive enhancement. *A List Apart*, October 2008b. URL http://alistapart.com/article/understandingprogressiveenhancement.

A. Gustafson and J. Zeldman. *Adaptive Web Design: Crafting Rich Experiences with Progressive Enhancement*. Easy Readers, 2011.

E. Hardy. Google adopts a new strategy: Mobile first, 2010. URL http://www.brighthand.com/default.asp?newsID=16235&news=Google+Android+OS+CEO+Eric+Schmidt+Mobile+First.

M. Hinchliffe. Responsive image placeholders, May 4, 2013. URL http://maketea.co.uk/2013/05/04/responsive-image-placeholders.html.

IETF. Charter for working group, 2014. URL https://datatracker.ietf.org/wg/httpbis/charter/.

D. Jackson. Improved support for high-resolution displays with the srcset image

attribute, August 12, 2013. URL https://www.webkit.org/blog/2910/improved-support-for-high-resolution-displays-with-the-srcset-image-attribu

S. Jehl, M. Marquis, and S. Jansepar. Picturefill documentation, 2014. URL https://github.com/scottjehl/picturefill.

B. Kitchenham, S. Pfleeger, L. Pickard, P. Jones, D. Hoaglin, K. El Emam, and J. Rosenberg. Preliminary guidelines for empirical research in software engineering. *Software Engineering, IEEE Transactions on*, 28(8):721–734, Aug 2002. ISSN 0098-5589. doi: 10.1109/TSE.2002.1027796.

K. Knight. Responsive web design: What it is and how to use it, 2011.

E. Law. Best practice: Get your head in order. *MSDN Blogs*, July 18, 2011. URL http://blogs.msdn.com/b/ieinternals/archive/2011/07/18/optimal-html-head-ordering-to-avoid-parser-restarts-redownloads-and-improve aspx.

A. Lumsden. All you need to know about the html5 data attribute. *Tutsplus*, November 29, 2012. URL http://webdesign.tutsplus.com/tutorials/all-you-need-to-know-about-the-html5-data-attribute--webdesign-9642.

E. Marcotte. Responsive web design. *A List Apart*, 306, May 25, 2010. URL http://alistapart.com/article/responsive-web-design/.

E. Marcotte. *Responsive Web Design*. A Book Apart, New York, 2011.

M. Marquis. Responsive images: How they almost worked and what we need. *A List Apart*, (343), January 31, 2012. URL http://alistapart.com/article/responsive-images-how-they-almost-worked-and-what-we-need.

P. McLachlan. Why domain sharding is bad news for mobile performance and users, October 30, 2012. URL http://www.mobify.com/blog/domain-sharding-bad-news-mobile-performance/.

P. McLachlan. Does mobile web performance optimization still matter?, January 30,

2013. URL
http://www.mobify.com/blog/web-performance-optimization/.

P. Meenan. How fast is your website? *Communications of the ACM*, 56(4):49–55, 2013.

G. Mineki, S. Uemura, and T. Hasegawa. Spdy accelerator for improving web access speed. In *Advanced Communication Technology (ICACT), 2013 15th International Conference on*, pages 540–544, Jan 2013.

Mozilla. Shorthand properties, 2013. URL https://developer.mozilla.org/en-US/docs/Web/CSS/Shorthand_properties.

A. Nicolaou. Best practices on the move: Building web apps for mobile devices. *Queue*, 11(6):30:30–30:41, June 2013. ISSN 1542-7730. doi: 10.1145/2493944.2507894. URL http://doi.acm.org/10.1145/2493944.2507894.

J. Nielsen. Response times: The 3 important limits. *Usability Engineering*, 1993. URL http://www.nngroup.com/articles/response-times-3-important-limits/.

B. J. Oates. *Researching information systems and computing*. Sage, 2005.

G. Pallis and A. Vakali. Insight and perspectives for content delivery networks. *Communications of the ACM*, 49(1):101–106, 2006.

G. Podjarny. Performance implications of responsive design, July 11, 2012. URL http://www.guypo.com/uncategorized/real-world-rwd-performance-take-2/.

G. Podjarny. What are responsive websites made of?, April 29, 2013. URL http://www.guypo.com/mobile/what-are-responsive-websites-made-of/.

M. Rouse. Definition – graceful degradation, March 2007. URL http://searchnetworking.techtarget.com/definition/graceful-degradation.

R. Sharp. What is a polyfill?, October 8, 2010. URL http://remysharp.com/2010/10/08/what-is-a-polyfill/.

S. Souders. *High performance web sites - essential knowledge for frontend engineers: 14 steps to faster-loading web sites.* O'Reilly, 2007. ISBN 978-0-596-52930-7.

S. Souders. High-performance web sites. *Commun. ACM*, 51(12):36–41, Dec. 2008. ISSN 0001-0782. doi: 10.1145/1409360.1409374. URL http://doi.acm.org/10.1145/1409360.1409374.

S. Souders. Domain sharding revisited, September 5, 2013. URL http://www.stevesouders.com/blog/2013/09/05/domain-sharding-revisited/.

B. Suda. *Creating Symbol Fonts.* Five Simple Steps, Penarth, UK, 2013.

The HTTP Archive. Interesting stats (desktop), November 15, 2010. URL http://httparchive.org/interesting.php?a=All&l=Nov%2015%202010.

The HTTP Archive. Interesting stats (desktop), February 1, 2014. URL http://httparchive.org/interesting.php?a=All&l=Feb%201%202014.

T. Theurer. Performance research, part 2: Browser cache usage – exposed!, January 4, 2007. URL http://yuiblog.com/blog/2007/01/04/performance-research-part-2/.

W. Tichy. Should computer scientists experiment more? *Computer*, 31(5):32–40, May 1998. ISSN 0018-9162. doi: 10.1109/2.675631.

R. Toy. Chrome 34 beta: Responsive images and unprefixed web audio, February 27, 2014. URL http://blog.chromium.org/2014/02/chrome-34-responsive-images-and_9316.html.

UVD Blog. Media query syntax, photograph, September 2, 2010. URL http://www.uvd.co.uk/blog/going-mobile-with-css3-media-queries/.

A. Vakali and G. Pallis. Content delivery networks: status and trends. *Internet*

Computing, IEEE, 7(6):68–74, Nov 2003. ISSN 1089-7801. doi: 10.1109/MIC.2003.1250586.

B. Venners. Orthogonality and the DRY principle – a conversation with andy hunt and dave thomas, part ii, March 10, 2003. URL http://www.artima.com/intv/dry.html.

W3 Techs. Usage of spdy for websites, April 2014. URL http://w3techs.com/technologies/details/ce-spdy/all/all.

W3C. W3 note process, 1996. URL http://www.w3.org/Consortium/Process/NOTE.html.

W3C. Hypertext transfer protocol – HTTP/1.1, June 1999. URL http://www.w3.org/Protocols/rfc2616/rfc2616.html.

W3C. Scalable vector graphics (SVG) 1.1 (second edition), August 16, 2011. URL http://www.w3.org/TR/2011/REC-SVG11-20110816/.

W3C. The picture element – an HTML extension for adaptive images, 2013a. URL http://www.w3.org/TR/2013/WD-html-picture-element-20130226/.

W3C. Use cases and requirements for standardizing responsive images, November 7, 2013b. URL http://www.w3.org/TR/2013/NOTE-respimg-usecases-20131107/.

W3C. The srcset attribute – an HTML extension for adaptive images, 2013c. URL http://www.w3.org/TR/2013/WD-html-srcset-20130228/.

WebPagetest. Webpagetest documentation - speed index, 2014. URL https://sites.google.com/a/webpagetest.org/docs/using-webpagetest/metrics/speed-index.

S. Weintraub. Industry first: Smartphones pass pcs in sales, 2011. URL http://tech.fortune.cnn.com/2011/02/07/idc-smartphone-shipment-numbers-passed-pc-in-q4-2010/.

Y. Weiss. Preloaders, cookies and race conditions, September 28, 2011. URL http://blog.yoav.ws/2011/09/Preloaders-cookies-and-race-conditions.

Y. Weiss, M. Marquis, and M. Caceres. A q&a on the picture element, March 17, 2014. URL http://alistapart.com/blog/post/picture-element-qa.

E. Weyl. Clown car technique: Solving adaptive images in responsive web design. *Smashing Magazine*, June 2, 2013. URL http://www.smashingmagazine.com/2013/06/02/clown-car-technique-solving-for-adaptive-images-in-responsive-web-design/.

M. Wilcox. Adaptive images documentation, 2012. URL http://adaptive-images.com/details.htm#how-it-works.

M. Wilton-Jones. Efficient javascript. *Dev.Opera*, November 2, 2006. URL https://github.com/operasoftware/devopera-static-backup/tree/master/http/dev.opera.com/articles/view/efficient-javascript.

L. Wroblewski. *Mobile first.* A Book Apart, New York, 2011.

J. Yuan, L. Xiang, and C.-H. Chi. Understanding the impact of compression on web retrieval performance. *The Eleventh Australasian World Wide Web Conference*, 2005. URL http://ausweb.scu.edu.au/aw05/papers/refereed/yuan/paper.html.

N. C. Zakas. The evolution of web development for mobile devices. *Queue*, 11(2): 30:30–30:39, Feb. 2013. ISSN 1542-7730. doi: 10.1145/2436696.2441756. URL http://doi.acm.org/10.1145/2436696.2441756.