Evaluation and Optimization of Web Application Performance Under Varying Network Conditions

Tomáš Černý[1]
cernyto3@fel.cvut.cz

Michael J. Donahoo[2]
jeff_donahoo@baylor.edu

Abstract: System performance testing measures application responsiveness for end users and aids in identifying bottlenecks in service delivery. For web applications, performance is typically reported in page load times, which are affected by many factors including caching, resource counts, etc. Providing complex UI functionality often greatly increases the number and complexity of page resources. We describe and demonstrate the use of a debugging web proxy, CZProxy, to allow analysis of the individual components of page load. Since network characteristics may significantly impact the true bottleneck, our proxy can simulate both latency stretch and bandwidth restrictions, allowing optimization over a variety of end-user connection scenarios. Using our debugging proxy, we conduct a case study on an actual web application, develop optimizations based on this study, and evaluate performance. From this, we provide a set of recommendation for web application optimization using a web debugging proxy.

Keywords: Web page load time optimizations, performance evaluation

1 Introduction

Web application development requires testing for both correctness and performance. Performance testing determines if the application is adequately responsive to various end user requests and, if not, identifies the most significant performance bottlenecks. Responsiveness in web applications is typically measured as the load time of various critical application pages. Pages are composed from a set of page resources, both static (e.g., image file) and dynamic (e.g., table generated from a database). Modern technologies like AJAX[1] or AHAH[2] may help with performance when only a partial fragment is being requested and embedded in the existing page, but also these technologies may experience its performance issues.

1.1 Elements of Page Load Time

User expectations of desktop-like interfaces in web applications continues to grow as demonstrated by modern web-based mail clients, Google Docs, etc. Dealing with these demands requires additional page resources such as JavaScript (JS), Flash, etc. Developers typically depend in part on third-party, component libraries, such as JQuery[3], to provide an advanced UI experience. Not only do such libraries further increase the number and complexity of page resources, but also these elements are usually not under the direct control of the developer. That is, page load time may be adversely affected by outside parties.

[1] Department of Computer Science and Engineering, Czech Technical University, Prague, CZ

[2] Department of Computer Science, Baylor University, Waco, TX, US

To locate opportunities for optimization, we need to know the load time and characteristics of the various page resources. To accomplish this, we use a web debugging proxy called CZProxy[4]. This proxy intercepts and relays all resource requests from the end-user's browser and records the time and total bandwidth needed to satisfy the request for each element. Developers can then observe the cost of various elements of the application and identify an ordered list of contributors to page load time. Once we identify the resources with the largest impact on page performance, we can explore various options for optimization, including decreasing computational complexity for dynamic elements, manipulating static resources, improved server and client-side caching, fetching strategies, etc.

1.2 Network Emulation

Application clients differ greatly in the available quality of service from the network. Web applications are deployed on one or a small number of servers at specific locations in the Internet topology. The location of a client relative to its server impacts end-to-end propagation delay. In addition, clients vary in the bandwidth quality of their network connection or specific bottlenecks in ISP peerings. High client latency and/or low bandwidth can significantly impact page load time. In fact, one may optimize page elements differently depending on anticipated bandwidth and delay. Unfortunately, the impact of network characteristics cannot be easily evaluated in local testing where propagation is small and bandwidth availability is high. In fact, local testing may fail to identify problems in page load time, which only manifest themselves under certain network conditions.

To evaluate the impact of varying network characteristics, CZProxy incorporates the ability to emulate various network characteristics, such as bandwidth restrictions and/or expanded latency. Testers could easily derive anticipated network characteristics using simple, widely-available measuring tools (e.g., ping). This allows testers to consider performance for a variety of clients. In addition, optimizations can be evaluated in a variety of network contexts.

The tool works by acting as a web proxy for the testing browser. The proxy can then observe/record all incoming and outgoing traffic, measure performance of various elements, and modify traffic to emulate network characteristics, etc. Based on information gathered during testing, application developers can experiment with a variety of optimizations under a variety of network conditions to tune the application to best server its client base.

1.3 Optimization

To develop optimization recommendations, we employ a case study, which evaluates an application with an international audience using CZProxy. From this, we identify the page resources creating the most significant bottleneck for application performance. We then propose and evaluate various optimizations. Much work has been done on database and service optimization [5][14][15], caching [6][7], etc. so we focus our efforts on resource configuration. Finally, based on the results of our optimizations, we propose a generalized set of guidelines for page resource configuration optimization.

1.4 Organization

We organize the rest of the paper as follows. In Section 2, we focus on our case study. We employ various techniques for optimizing load time of selected pages and provide an explanation for each procedure. Section 3 presents our generalized optimization guidelines for page resource configuration. In Section 4, we deal with related work in this area. Finally, we present relevant conclusions and future work.

2 Web Application Case Study

For our case study, we examine large enterprise web application for contest management (CM)[11] used by ACM International Collegiate Programming Contest (ICPC) organization. We show how to evaluate and improve the application web page load times using our web debugging proxy. We discuss optimizations and improve the page load time step by step with provided measurements. For our study, we choose three structurally diverse web pages of the application: *Dashboard* (user summary information from a variety of sources), *Contest* (evaluation of a very large page), and *Team* (information about competing teams).

We also build a table of approximated bandwidth and propagation delay for remote locations to evaluate remote application performance. We summarize how the optimization influences page load time in distant locations at the end. The application we evaluate is J2EE application using JBoss Seam[8] and RichFaces[9] components.

2.1 Network Characteristics

In this section, we develop an approximation of network characteristics for remote locations. Latency and bandwidth are the main factors in our measurement. To measure these factors, we attach CZProxy to a web browser with clean cache and navigate to a particular location. The tool captures all the information about requested resources. These do not always request only a single location, but also Google, Amazon, commercials, etc. In the first stage, we measure the network parameters using our tool, then we statically download the tested web page and run locally with the network parameters applied such that the local static page behaves similar to the remote one. We particularly focus on static fragments like cascading style sheets (CSS), JS, or images so that we reduce the server processing time impact for dynamic fragments. The measurement is made from CTU, Prague. We measure 4 samples and provide the averaged in Table 1.

	Measured by CZProxy				Simulation parameters		
Location	Bandwidth [kb/s]	Shortest request [ms]	Resources	Size [kB]	Bandwidth [kb/s]	Latency [ms]	Location URL
Japan	251.11	657.5	83	850	251	600	japantimes.co.jp
China	886.35	904	116	1812	886	800	ebeijing.gov.cn
Germany	2653.62	56	154	1248	2653	45	spiegel.de
Russia	576.09	131.25	178	994	576	110	pravda.ru
Brazil	419.72	520	84	571	419	520	www.embratur.gov.br
USA	915.19	260.25	133	1307	915	250	www.cnn.com
Canada	541.12	306.25	110	668	541	260	www.cbc.ca/news
Australia	359.03	294.5	155	714	359	260	www.news.com.au

Table 1: Approximation of remote locations - parameters

The results for the Japan page are the following: 83 resources, size 850kB, load times range in 24196ms - 32270ms, avg. bandwidth 251.11kb/s, the largest resource for loading 11552ms and the avg. shortest request took 657.5ms. In the measurement, we received values for the bandwidth in range 217kb/s to 297 kb/s and shortest response times in range of 561ms to 788 ms. Network conditions significantly change with the time, because of random traffic congestion from Czech to Japan. We approximated this site locally with proxy setting 251kb/s for max. bandwidth and 600ms for minimal latency. The traffic fluctuation means that the simulation of the remote page will be under average conditions between a Czech client and Japan server. We evaluate the other web pages in the same way.

2.2 Initial Page Load Performance

In our study, we focus on optimizing our case study application. We find the bottlenecks that impact the entire page load time, via the attached proxy. The browser cache is erased to eliminate its impact on the measurement. As mentioned earlier, we use the CM pages *Dashboard*, *Contest* and *Team*. At the later point, we also pick a Contest *Standings* page to show how a simple feature can affect the entire load time.

2.2.1 Initial measurement

Our first look is at the unoptimized web application. The primary parameter is the number of requested resources. The log shows that the majority of requests are for pictures, CSS and JS files. Each element shows its content and size. The most importation parameter that is influenced by the previous two is timing. CZProxy shows both timing per request and also overall time to load the page. Every experiment is made five times with clean browsers cache and with cached resources. Table 2 provides the results.

	Time[ms]/cached	Size [kB]/cached	Resources [-]	Japan - Time[ms] /cached	Canada - Time[ms] /cached
Dashboard	2504/2437	907/805	79	30145/27722	14435/13434
Contest	11019/10557	1657/1553	97	59706/56518	29574/28656
Team	3439/3191	1339/1225	117	48207/46181	24689/23164

Table 2: Initial time to load the page with approximated location in the Japan. and Canada

The initial load time for the Dashboard page is 2504ms, its size is 907kB and it consists of 79 resources. When we hit the page for the second time, we see that browser cached few resources, and the load time and load size decreases only marginally. The contest page takes the longest time to load, which is not only due to its size, but also because of its complexity at the server side, the dynamically generated main fragment. On the other hand, the Team page has the largest number of resources to compose the web page.

We received a result of the measurement from almost ideal conditions. The network connection is excellent because the client and server are on the same workstation. Next we consider network characteristics for some remote locations. For our positions in Prague, we apply settings for Japan (bandwidth 251 kb/s, latency 600 ms) and Canada (bandwidth 541 kb/s, latency 260 ms). From the Table 2, we see that the application load time is significantly slowed down, and we almost cannot use the CM application in Japan.

2.3 Optimizations

To improve the load times, we apply optimizations and measure how the particular optimization influences the page loads. For each optimization, we explain what is the purpose and discuss the results. Our default network connection in this study is a local-area connection with high bandwidth and minimal propagation delay. To evaluate improvements for more realistic network conditions, we emulate various network characteristics using our proxy and evaluate the impact of the changes.

2.3.1 Obfuscation

All pages contain many static resources. This includes text files, which contain significant whitespace thereby increasing overall size; however, such whitespace is only for human readability. To address this, we apply obfuscation, which takes off all unnecessary white spaces and comments. The tool we use for obfuscation is YUI Compressor [10].

	Time[ms] / cached	Size [kB] / cached	Resources [-]
Dashboard	2149/2082	**712/642**	79
Contest	11324/10586	**1354/1281**	97
Team	3384/3149	**1016/935**	117

Table 3: Time to load the page after obfuscation

By applying this method we reduced CSS and JS files sizes from 15% to 50% (files that we developed). From the Table 3. we see that we reduced the page size by 200 to 300kB.

2.3.2 Merging

The next improvement is a merge of CSS and JS files, which decreases the number of resources. If we have 10 CSS files, the browser has to issue 10 separate requests and receive 10 separate responses. Each connection suffers overhead from the TCP handshake and slow start. **Merging** reduces this to one request and one response. In addition to CSS and JS files developed specifically for the CM application, CM utilizes a third-party library called RichFaces[9] that provides an option to merge all available CSS and JS resources. The merge contains all resources, even those not used on the requested page. This provides an opportunity for pre-fetching if the browser cache is used for the resource. If we use both RichFaces CSS and JS merge options, it naturally increases the total web page size. We provide two measurements one for all resources merged and one where Rich Faces JS files are not merged.

	Time[ms] / cached	Size [kB] / cached	Resources [-]		Time[ms] / cached	Size [kB] / cached	Resources [-]
	JS and CSS library merged				*CSS library merged*		
Dashboard	9675/9551	1360/1293	**36**		2264/2123	773/703	**61**
Contest	13398/13375	1727/1656	**41**		11059/10842	1397/1326	**74**
Team	10834/10785	1337/1256	**52**		3513/3434	1051/970	**91**

Table 4: Page load times with RichFaces JS and CSS library merged

Note that our page load times increased even though the number of resources dropped significantly. So why to make such an optimization? First of all, the page load is slowed down strictly due to the size of the page. Second the latency in our experiments is very small; however, load time increases quickly with latency increase because each and every resource request is impacted by latency. In the next step, we enable the resource cache, so the initial load of the application loads all the necessary resources and all the following page requests use the JS and CSS resources from the cache, which provides some pay-off.

2.3.3 Allow the Browser Cache

The pages contain a significant number of icons. We could reduce them in one large image and use CSS to display a sub-position of the image. Or we can apply **cache** for a significant amount of time. So the user will load them just once. We may apply for CSS and JS.

	Time[ms] / cached	Size [kB] / cached	Resources [-]
Dashboard	9675/**2044**	1360/**260**	36
Contest	13398/**12323**	1727/**623**	41
Team	10834/**3395**	1337/**227**	52

Table 5: Page load time after caching applied (JS and CSS merged)

The result table looks better for the cached pages, but we should keep in mind that the initial load time happens just once if we go with the option to load all merged JS files. The JS pack is around 930 kB (uncompressed). There is a marginal improvement for Contest page. If we look at timing at our proxy, the main bottleneck is the dynamic content (613kB). The debugger allows you to see which resources are cached. Non-cached lookups have the response header *200 OK* on the other hand response *304 Not Modified* is used when cached.

2.3.4 Compress the Data Transfer

All the previous measurements had one thing in common. All of the pages were quite large, so what should we do is to decrease the size of transferred data. This can be simply done by **compression** of the resource content.

	Time[ms] / cached	Size [kB] / cached	Resources [-]	Time[ms] / cached	Size [kB] / cached	Resources [-]
	JS and CSS library merged			*CSS library merged*		
Dashboard	**1716/1269**	**319/26**	36	**1096/1042**	**176/32**	61

Contest	**5205/4625**	**373/78**	41	**4691/4401**	**294/84**	74
Team	**3347/3062**	**340/37**	52	**2996/3114**	**279/45**	91

Table 6: Page load time with applied compression - RichFaces JS and CSS library merged

The page load dropped significantly because of the page size. If we do not merge the JS library, we might get better results but more resources. It is hard to claim that one option is better than the other. From Table 6, it may seem that not pre-fetching the JS is a better option. But later we see results for the Japan and Canada.

2.3.5 Remote Clients

Next we consider emulation of latency and bandwidth for client location in Japan and Canada. The results are provided in Table 7.

	Japan - Time*[ms]* /cached			Canada - Time*[ms]* /cached		
	Original	*CSS library merged*	*JS & CSS lib merged*	Original	*CSS library merged*	*JS & CSS lib merged*
Dashboard	30145/27722	**6861**/1778	11365/**1747**	14435/13434	**3302**/**1483**	5952/1542
Contest	59706/56518	**10818**/5458	13679/**4939**	29574/28656	**6738**/4758	7112/**4505**
Team	48207/46181	**10129**/4724	11942/**4307**	24689/23164	**4986**/4166	6312/**4077**

Table 7: Page load time [ms] in the Japan and Canada using optimizations (CSS and JS pre-fetch)

To load the Contest page in Canada now takes 4.4 times less time (6 times with cache). In Japan the load time decreased 5.5 times (10.4 with cache). We can see that to pre-fetch JS and CSS library is reasonable when we expect the user to spend more time in the application.

2.3.6 Feature Evaluation

Testing the impact of a view feature may be also very useful for recognizing the bottleneck in the application. In the Standings page, we disable a feature that abbreviates text in a table column and displays the full length if the abbreviated element is hovered (results in Table 8).

Standings rows (old)	Time[ms]/cached	Size [kB]/cached		Standings rows (new)	Time[ms]/cached	Size [kB]/cached
1000	4270/4733	200/88		1000	939/929	163/47
70	1097/1034	134/18		70	540/548	131/15

Table 8: Feature evaluation

Considering the compressed data, the feature seems to be quite expensive. Standings sometimes contain up to 2000 records, and this undoubtedly impacts load time. We reconsider the use of this feature because of timing issues.

3 Generalized Optimizations

To begin web application optimization, we separate the task in two different parts. Backend optimization is concerned with algorithms, server caching, SQL optimization etc. In this paper, we focus on frontend optimization. We use CZProxy to receive timing for each request, to request content, and get dependencies.

3.1 Find The Bottleneck

We use the proxy to find the application bottleneck at the server side. We request a simple test page of the web application under evaluation. In the next step, we request a target page and follow the time response difference between the simple test page load and the one we are concerned. If there is a significant difference, the page rendering is the bottleneck. This can be for two reasons: 1) we are generating too much HTML code or 2) the rendering process takes too long. We should debug the rendered page, disable page segments, and retest the load times. This way we find the segment that causes the bottleneck. The bottleneck part might be caused by a complicated aggregation, SQL query, or too much unoptimized HTML with repetitions. Too much HTML can be found in the proxy reported response content.

3.2 Optimize The Page Data Transfer

If the requested page is large and accessed over small bandwidth or large propagation delay, you should enable page compression, which is normally done by web server if configured properly. In this case, the content transferred from server to client is compressed. The compression and decompression takes some overhead, but the transfer data size reduction has a predominant influence.

3.3 Use Web Browser Cache

Static resources like images, CSS or JS do not change often and most likely will not change for days or months. It is reasonable to cache them at the web browser by setting the page header *pragma max-age*. Some web servers allow this setting in its configuration for given resources.

3.4 Reduce White Spaces - Obfuscate

Static text content (CSS, JS) contains significant whitespace and comments. There is no need to send these, because it does not have any meaning. For the page rendering, you can obfuscate static text content by YUI Compressor. Reduce image size by a lossy compression.

3.5 Reduce Resources - Merge

If you have hundreds of static resources that are used for a page rendering, you need to request every piece individually from the server. This means that there will be hundred of requests and responses. If all hundred resources are merged in one, then there is only one request and response, greatly reducing overhead. Reduce as many resources as possible, although most of the resources are fetched in parallel. Avoid inlining of the code to every resource to reduce the total size. Unfortunately, there in no easy way to pipeline pictures. If there is many icons in your application, you may merge them in one picture and use a CSS to request a position of the given icon in the picture `(background: url('iconMerge.png') 50px 20px no-repeat)`, reducing most of the image requests.

3.6 Disable Unnecessary Features.

It is always desirable to provide the user with nice features that makes the web application feel like a standalone one, but it has also a down side as it might produce extra code that needs to be rendered and it may take time to generate and submit the page. You might find the "trouble" features by the proxy when commenting out page fragments and test the page timing.

4 Related Work

Optimizations apply in many areas of enterprise application development. Much work has been done on database and service optimization [5, 14, 15], caching [6, 7], etc. Our focus in this paper is resource configuration and its impact on web application page load time. CZProxy is an open-source tool project. We discuss the details of this tool in [4]. The related applications mostly focus on debugging and provide no guidelines for performance optimization. Our tool predates some more recent projects. One of them is Page Speed[12]. Its user documentation provides very rich guidelines for optimizations where they also focus on browser rendering. Unfortunately, this tool cannot adjust network conditions. Another very interesting idea comes from Google labs called Webmaster Tools[13]. A developer can add a public web project URL and these tools help him with search result performance evaluation, also all hacker attacks can be identified. These tools also provide the developer with timing, latency and the average page load time information about the site. The user documentation provides reasonable set of recommendations for the developers. The disadvantage is the requirement for publicly accessible web applications and provided feedback from one location and not multiple.

5 Conclusion and Future Work

In this paper, we show how to identify bottlenecks in web applications. In addition, we provide a tool called CZProxy that helps with such an evaluation. Next, we evaluate a real enterprise application case study and significantly improved web page load times. We even received anecdotal feedback from China that the application speed significantly improved after the optimizations. The study shows common optimization techniques. There are two strategies of web application configurations: 1) minimize the number of transferred files, which may increase the total page size or 2) there are more files are to transfer, but with smaller total size. We recommend using the first strategy for web applications where we expect the user to navigate among the application and fill in some data. The second is better for News servers, where the user wants to see the main page as fast as possible.

The main contribution of this paper is to support the knowledge and guide to optimize web applications from the resource configuration perspective. There are many web applications world wide that could benefit from our recommendations. All end users appreciate faster page loads and less congested Internet.

References

1. AJAX, Asynchronous JavaScript and XML, http://www.w3schools.com/Ajax/
2. AHAH Asychronous HTML and HTTP, http://microformats.org/wiki/rest/ahah.
3. JQuery, JavaScript Library, http://jquery.com.
4. Tomas Cerny, Michael J. Donahoo, A Tool for Evaluation and Optimization of Web Application Performance, MOSIS 2010, binary version, http://cz-proxy.wiki.sourceforge.net.
5. An Overview of Query Optimization in Relational Systems, Surajit Chaudhuri, Microsoft Research, 1998.
6. Caching, A survey of web caching schemes for the Internet, Jia Wang, Cornell University Ithaca, NY, ISSN:0146-4833, 1999.
7. A Web Caching Primer, Brian. D. Davison, IEEE Educational Activities Department Piscataway, NJ, USA, ISSN:1089-7801, 2001.
8. Seam Web Application Framework, http://www.jboss.com/products/seam.
9. RichFaces User Interface Library, http://www.jboss.org/jbossrichfaces.
10. YUI Compressor, http://developer.yahoo.com/yui/compressor.
11. Contest management system (CM), ACM ICPC. http://cm.baylor.edu.
12. Page Speed, http://code.google.com/speed/page-speed.
13. Webmaster Tools, http://www.google.com/webmasters/tools.
14. SQL Query Optimization: Reordering for a General Class of Queries, Piyush Goel, Bala Iyer, SIGMOD ’96, Montreal, Canada.
15. SQL Query Optimization through Nested Relational Algebra, Bin Cao, Antonio Badia, ACM Transactions on Database Systems, Vol. 32, No. 3, Article 18, Publication date: August 2007.
16. A Tool for Evaluation and Optimization of Web Application Performance, Baylor University Technical Report, 2010.