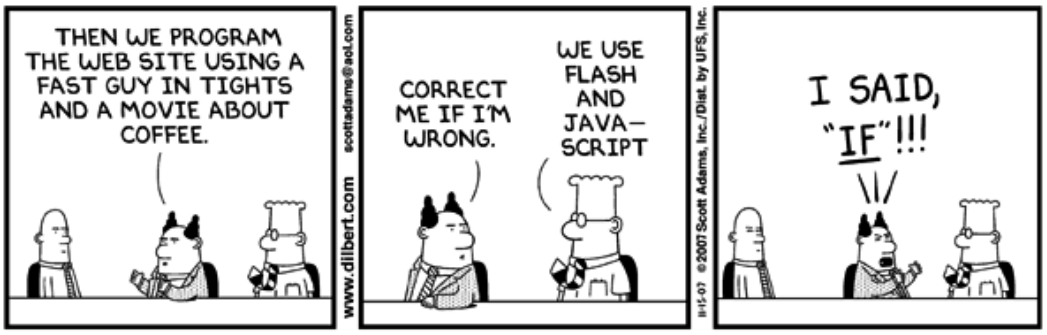
Continuous Live Information Monitor and Benchmarking

A JavaScript Library For Client Benchmarking and Code Modification



Master's Thesis by
Morten Bøgh
Markus Krogh

Aalborg University, June 12th, 2009

Department of Computer Science
Aalborg University
Selma Lagerlöfs Vej 300
Phone 96 35 80 80, Fax 98 15 97 57
http://www.cs.aau.dk

Title:
Continuous Live Information Monitor and Benchmarking
- A JavaScript Library For Client Benchmarking and Code Modification

Theme:
Programming Technology

Project period:
DAT6, spring 2009

Project group:
d625b

Authors:
Morten Bøgh
Markus Krogh

Supervisor:
Kurt Nørmark

Printcount: 4

No. of pages: 59.

Appendixcount: 3

Completed:
12-06-09 at Aalborg University.

Abstract:

In recent years web site development has gone toward creating Rich Internet Applications powered by JavaScript. Rich Internet Applications are created to improve the user experience and provide interesting features. With the recent improvement of web browsers and JavaScript execution, developers can make advanced Rich Internet Applications. However with the popularity of mobile devices such as phones and netbooks, these Rich Internet Applications have become a problem. As it is now, the developer must choose between the strong or weak clients.
This thesis sets forth five hypotheses, which states that it is possible to aid developers by creating a JavaScript library which continuously benchmarks the client and is able to modify existing JavaScript code accordingly. To confirm these hypotheses, a prototype of the proposed library has been implemented. The library is called Continuous Live Information Monitor and Benchmarking (CLIMB), and through a series of tests some of the hypotheses were found to be true. In order to confirm the remaining hypotheses, the library needs a final touch and a real life test case.
This thesis shows, that it is plausible, that a library such as CLIMB is part of the solution for the weak clients.

Contents

Preface

This master's thesis has been written during the 2009 Dat6-project period, by computer science group d625b at Aalborg University. The main theme is "Programming Technology". It is addressed to other students, supervisors and anyone else who might be interested in the subject. To read and understand the report correctly, it is necessary to have knowledge about the most basic computer related terms and web development.

Abbreviations and acronyms will at first appearance be written in parenthesis, to avoid breaking the reading stream. Specification of gender in the report is not to be understood as suppression or any form of political/religious position. The gender is only specified to simplify the process of writing for the authors.

References to sources are marked by [#], where # refers to the related literature in the bibliography at the end of the report.

The appendix to the report is found in the last chapter of the report. All source code created during the project is located on the following url: `http://boundless.dk/climb/`. An electronic version of this report in pdf format can also be obtained from that url.

We would like thank the F-Klub at Aalborg University for bringing light, and occasional breaks to our project period.

Chapter 1

Introduction

In recent years web site development has gone toward creating richer Internet applications, where the user experience is in focus and interaction is favored over static web sites. This tendency is called Rich Internet Application (RIA) and it is all about giving the user a better experience while visiting the web site/application, added functionality, and a more responsive Graphical User Interface (GUI), as is shown by Kevin Mullet in *The Essence of Effective Rich Internet Applications*[1]. In our previous work *Unified Multitier Web Development*[2] the analysis showed that an increasing amount of web development frameworks focus on RIA development.

The richer experience was originally created with Adobe Flash[3], but today it can be achieved using JavaScript (JS), both for visual manipulation and server interaction. With JS, the developer can create web sites and web applications that have similar responsiveness as is known from desktop applications. This responsiveness is mainly achieved with Asynchronous JavaScript and XML (AJAX), where the client's interaction with the server is hidden from the user, thereby making the web site appear more interactive.

As RIAs become even richer, the use and amount of JS grows. This implies that more and more work is moved to the client, e.g. sorting data in tables or performing various computations. This raises the requirements for the client, and in 2009 there are still many weak clients on the Internet and as mobile phones are making their way onto the web the number will only increase. The growing popularity of browsing via the mobile phone is a problem for the developer. Although mobile phones are becoming more and more powerful, they are still weak compared to a standard computer today (2-core system with 2 GB ram). Thus the higher requirements for the client is a problem, a problem that is non-trivial to solve.

Rich Internet Application developers are facing these problems right now and currently they have to decide which group of clients to target.

This report proposes a light weight JS library, which continuously performs client benchmarking and which is able to change existing JS code to fit the condition of the client in order to solve the problems described above. Network status is also considered, as server side interaction is a large part of RIAs. This will therefore be placed in the benchmark part of the library. Modifications done to the JS can be done in two ways, rewriting a function completely or inserting code before and/or after the function in question. The before/after concept is inspired from the programming paradigm Aspect-oriented programming (AOP),

which also operates with the before and after concept, although there is much more to AOP than just this.

The developer can use the library in different ways, he can import it to his project and use it as it is, which will enable benchmarking of the client and modification of commonly used features such as the `XMLHttpRequest`. Another usage of the library is to create application specific rewriting by creating modifications which target selected areas of the application. An example could be movement of an object from point A on the screen to point B, the original function would move the object across the screen in a fluent motion, an alternative version of the function could just move the object from point A to point B in a single step. The user can use the monitor as a service tool, as it can be used to give him information about his system and the performance he can expect from a RIA. From this point the monitor will be referred to as Continuous Live Information Monitor and Benchmarking (CLIMB) and will be described in more detail in section 4.1.

The main contribution of this thesis is a light weight JS library(CLIMB), which benchmarks the client and automatically changes the code either based on predefined or custom made filters. CLIMB could ultimately be used to allow dynamic placement of execution in web applications, that is moving running computations between the client and the server. CLIMB, as developed in this thesis, is a tool that enables developers to develop performance dynamic applications that automatically adapt to the current platform.

This thesis also provides insight in JS co-existence, i.e. how to make JS applications co-exist, which is elaborated in section 2.2.3.

The knowledge obtained from developing CLIMB, which is communicated in this thesis, is useful to people in the scientific community who are interested in web application performance design. CLIMB is interesting, as it takes a slightly different approach to monitoring than the existing solutions do. Where the existing monitors are meant to be used strictly for debugging, our solution is meant to be used live. It uses the information gathered from benchmarking to alter the behavior of JS code made by the developer. What to be altered is either defined by the built in filters or by filters created by the developer.

Our vision for CLIMB is, that it could advance RIA development by allowing older computers and mobile phones to run heavy RIAs.

The following chapters will try to solve these problems. This includes development of the JS library CLIMB, evaluation thereof, and lastly a conclusion of this project.

The remainder of this chapter introduces the core aspects needed for this thesis, related work, and our vision. These core aspects are to be found in the current web market, which has changed significantly over the last few years. The main contribution to these changes are the web browsers, the trend in web development, and the increase of client devices.

Related work to this project is introduced and their work is compared to what we are going to create. Our main vision is then presented to show how we believe we can aid RIA developers.

The remainder of this thesis will take the subject further and chapter 2 will analyze how benchmarking of browsers is done by the browsers vendors, and take a look at JS and the interesting aspects of it, in the light of this project. The existing monitors first presented in section 1.2 will be analyzed further to gain more insight into these projects.

Following this is the problem statement in chapter 3, which states five hypotheses which all deal with RIAs, and how to optimize them to assist weaker clients. Chapter 4 presents the development process and explains the decisions made prior to the development.

The implementation itself is presented later in chapter 4, where selected features of the implementations are explained. Problems during the development is also brought up. Chapter 5 is where CLIMB is tested and an initial answer for the hypotheses is given.

Evaluation of the whole project is done in chapter 6 and lastly the whole project is summed up in the conclusion in chapter 7.

1.1 Trends

As has already been mentioned the whole web market is experiencing change. This change can be seen in both web sites and web application experience, namely the functionality and appearance. In order to ensure a proper overview of the market, this section will cover the trends within web development, browsers and client devices.

Rich Internet Application

Rich Internet Applications have already been introduced earlier in this chapter and also analyzed in our previous work *Unified Multitier Web Development*[2]. Rich Internet Application is the trend in web development. So RIA is all about GUI, responsitivity, client independency, distribution and complex data handling. It all started with Adobe Flash, which gave the developers and designers the opportunity to create responsive and interactive GUIs. Flash however has its limitations, since it requires the installation of a plugin, and also compared to JS it often uses more resources even for simple animations. In recent years JS has taken over the position that Flash had in the RIA market. This is mainly due to two things. The first: JS works out of the box, all major browsers have support for it, and second: several JS libraries have emerged, such as jQuery[4], Prototype[5]. Frameworks such as Google Web Toolkit (GWT)[6] and Echo Web Framework[7] have made the development of RIAs much easier for developers compared to writing core JS.

Web Browsers

The web browser market has changed a lot in the last 2-3 years. Microsoft's browser Internet Explorer (IE), now in March 2009 at version 7, is still the most used browser, but it is slowly losing market shares to the competitors Mozilla, Opera, Apple, and now also Google, who is currently claiming market shares from IE with their new browser Chrome. Until recent IE's competitors competed on customizability, additional features, browser speed, and security, but lately the focus has shifted towards JS execution speeds. Mozilla was the first to announce a new JS engine called TraceMonkey, which will be shipped with Firefox 3.1 and it will boost the JS performance drastically. The next to focus on JS was Google, which until that point was not a player on the browser market. They released Chrome with a JS engine called V8, that outperformed Mozilla's unreleased TraceMonkey. Apple is the latest to announce a new and improved JS engine in the release of their browser Safari 4 Beta and this engine is supposedly faster than all the competitors. As all the competitors are focusing on faster JS execution speeds, Microsoft is going down a different path. A spokesman from the IE8 developer team has said that they focus on the overall user experience and with the release of the tech report *Measuring Browser Performance: Understanding issues in benchmarking and performance analysis*[8] they make it clear that they focus on the whole browser instead of "just" JS execution speeds.

Client Types

Traditionally a computer was the most common way to access a web site. Mobile phones were not powerful enough to run a full feature browser and the usage of internet data on a mobile phone was rather expensive. Vendors such as HTC and Apple have made the smartphone available to the main consumers. With their phones running Windows Mobile and OSX respectively, they have full featured browsers, which enables them to browse the web like a normal computer. Opera has made a mobile edition of their browser called Opera Mini, this browser is targeted lesser capable phones. In order to make browsing possible for these clients, the browser uses a special proxy, which means that all traffic is routed through a server at Opera, which has taken a "photograph" of the requested web page and compresses it. This however disables RIA as it is only a picture of the web site the user is presented with.

1.2 Related Work

This section is a short introduction to, what have been done in the field of both web monitoring and benchmarking. Both benchmarking and web monitors will be examined closer in chapter 2.

Web monitoring is a broad concept which includes several aspects, e.g. security, performance, and user interaction. These are all interesting aspects that have been explored to some extend within the area of web applications. A monitor is normally running alongside an application gathering data about said aspects in the application. The following will revolve around monitoring in JS.

A monitor to track user interaction was developed by Atterer et al.[9]. They have implemented it using a HyperText Transfer Protocol (HTTP) proxy, which injects JS into web pages before they are served to the client. The injected JS can report a wide range of information, from the users mouse movement to keyboard input. This information is then sent back to the proxy server where it is logged. Developers can then analyze this information in order to identify e.g. usability problems.

Emre Kiciman and Benjamin Livshits have presented a debug monitor, *AjaxScope*[10], which is based on the idea of the HTTP proxy devised by Atterer et al.[9]. It enables a developer to gather data on a web application, such as infinite loops, memory leaks, but it is also possible to do performance profiling. AjaxScope can be a valuable asset for maintaining deployed web applications, where you normally do not have much debugging information. In order to reduce the load on the client, AjaxScope distributes the test cases over several clients, this way only a small overhead is added to the client. To do this the HTTP proxy server remembers each client and can send new test cases to the client each time it uses the web application. AjaxScope allow developers to get instant feedback on modifications applied to the web application, since web applications are instantly deployed to all users. This means that AjaxScope will always collect information on the newest version of a web application.

Inspired by the work of Sands et al.[11] our monitor CLIMB is to be implemented as a lightweight JS library. The project most similar to this thesis is AjaxScope, since it allows a developer to do performance profiling. The information gained from AjaxScope can however not be utilized directly in a running application. It differs from CLIMB mainly in the way it is implemented and utilized. Where AjaxScope is implemented as a proxy server that injects JS, CLIMB is, as mentioned, a library, which is present during development. This

means that where AjaxScope only gather information for deployed web applications, with the intention of debugging, CLIMB can provide runtime information that can be used in the application, and not just for debugging. The idea behind CLIMB, which is described greater detail in section 1.3, is that the information gathered during runtime can be used to influence the web application. An example could be that a web application running on a computer, which has lost its network connection. It could then use CLIMB to detect and react to the lost connection. Such a scenario would not be possible in AjaxScope.

1.3 Vision

The creation of RIAs has become easier in the recent years, as more and more web development frameworks have come with support for it. Frameworks such as GWT[6], Echo Web Framework[7], Silverlight[12] etc. focus solely on developing RIAs and provide many features for enriching the user experience. These frameworks were analyzed and tested thoroughly in our previous work *Unified Multitier Web Development*[2]. They do however not provide any measures for the developer, that allows him to handle situations in which weak clients experience poor performance.

The vision of this project is to advance the development of RIAs by providing developers with a library for handling different types of clients. This library will help developers by tuning the user experience, based on information gathered from the client. It is hypothesized that this information should make the developer better equipped to handle the increasing amount of weak clients, without compromising the richness of the web application for normal clients.

The next step is how to realize this vision and the following chapter will go deeper into the existing web monitors, analyze JS and the use of JS to benchmark browsers.

Chapter 2

Analysis

In this chapter we first present a short introduction to benchmarking in general and then how JS benchmarking is done today by the browser vendors. This will give an indication of methods used to benchmark JS capabilities. These areas are of interest, as the further work in this report requires this knowledge of benchmarking. As the library described in the vision section 1.3 and in order to follow up on this we then describe JS, how it is implemented, what limitations it has and the specialties of JS.

When the basics have been covered, it is time to take a closer look at the JS monitors mentioned earlier in section 1.2. Based on those, it is possible to define how our performance monitor should be integrated in existing web applications.

2.1 JavaScript Benchmarking

Benchmarking is the process of testing an object, such an object could be a piece of software, which the developer wants to test for specific things regarding performance. Besides testing of software, it could be hardware like the CPU, RAM etc. In our case we are interested in benchmarking with respect to the client in terms of software(web browser) and the hardware, because it influences the performance of a RIA.

We define a benchmark as the following: *A set of tests to measure the performance of an application/hardware*

A benchmark can be seen as a *What* and a *How*. What do we want to test on the subject and how do we do it. The following two sections will cover the *What* and the *How* in the context of browser benchmarking and are based on the writings of John Resig[13], the creator of Dromaeo[14] JS benchmark suite and the light weight JS library jQuery[4]. He is currently working at Mozilla[15] with JS.

What

As mentioned it is important to determine what it is the benchmark should test and when it comes to web browsers and JS, it can either be JS language performance or Document Object Model (DOM) manipulation. This means that there are basically three ways to benchmark the JS performance of a browser.

JavaScript Benchmarking on the core of JS, shows how fast the JS engine of the browser is. A benchmark within this category usually involves ray tracing, cryptography, regular expression. These tests correspond to testing the execution speed of any other language.

Document Object Model Benchmark tests focusing on the DOM gives information about how the browser and the JS engine collaborate. The "DOM performance" is actually quite important in RIA as most of the work done by JS is DOM manipulation.

Combined A combination of the above two mentioned methods is preferable, as this gives a better picture of the browser/client performance. Both the JS and DOM performance are important when creating RIA.

All three ways are being used in existing JS benchmarks and these will be presented later in section 2.1.1.

How

The second question asks how to perform the benchmark and this is important since the method used in the benchmark will affect the accuracy. Basically there are two approaches on how the tests can be run, either the number of runs is limited or the amount of time is limited. Both approaches are used when benchmarking JS in web browsers.

If the number of runs is the limited factor, the runtime of the tests has impact on the number of runs needed to get accurate results. An example could be if a test takes a 1-2ms to run and the test is only run a few times, for this example 5-10 times, then small deviations in the individual result will have a large impact on the final benchmark result. These deviations could be a temporary disturbance from other applications running on the computer and thereby cause high error levels in the benchmark results. If it was not a library written in JS and run in the browser, which sandboxes the JS. It would have been possible to access the system on the client and check the actual process time given to the benchmark test and thereby take note of disturbance from other applications. This disturbance can be avoided in two ways, either by increasing the number of runs which will make the error level drop or heighten the runtime of the test. Heighten the runtime of the test will increase the runtime of each test run and rule out deviations caused by temporary disturbance and gain an accurate result.

There is however a problem with this way of constructing the benchmark, the browsers are becoming faster all the time and especially JS execution is becoming faster, as mentioned earlier in section 1.1. This speedup in JS execution means that the number of runs chosen, when the test was created, quickly will become insufficient, as a slow running test becomes faster and a fast running test becomes even faster and more error prone. An easy solution is to increase the number of runs over time as the JS execution becomes faster but this task of updating the test every single time a new version of a browser is released is too time consuming. Older computers and those who use old versions of a browser will not experience the speedup and then the runtime for the whole benchmark will grow to an unacceptable size in our case where the time factor is crucial.

The other approach and a more permanent solution to this problem, would be to limit the time a test is allowed to run and thereby rule out the problem of the advancement in JS execution speed. A test could e.g. be set to run for one second and then the benchmark result would be based on the number of iterations the test can be run within the one second

time frame. Freezing the time and not the number of runs also gives certainty about the amount of time the whole benchmark will take. This property is very useful when the benchmarking is used on the runtime of a web site, as the load caused by the benchmark can be controlled.

Now that the different aspects of benchmarking in the browser field has been analyzed, it is now relevant to examine the existing JS benchmark suites.

2.1.1 Existing JavaScript Benchmarks

Benchmarking of browsers is becoming a big deal, as they become uniform feature wise, such a big deal that some of the major browser vendors have made their own browser benchmarking suites. They use it for marketing by optimizing their browser for their own benchmark and the other benchmarks, thereby scoring better than the competitors.

There are three major JS benchmarking suites:

- Dromaeo by Mozilla[14].
- SunSpider by Webkit[16].
- V8 Benchmark Suite by Google[17].

These three benchmarking suites will now be analyzed and classified according to the *What* and *How*, which were described previously in this section.

Dromaeo

Dromaeo is a benchmark suite made by Mozilla, who is also behind the popular web browser Firefox. It was created by John Resig and currently he is the one maintaining it. Dromaeo uses a wide variety of test and of the benchmark suites in this analysis, it is the only one covering all aspects of *What* and *How*. So it has tests which are slow and fast in execution and they focus on both DOM and JS language performance. It uses a semi-fixed number of runs for each test, meaning that each test will be run five to ten times. The semi-fixed number of runs, works like in the following manner: If the first five runs show poor performance, then run six to ten will be executed in order to determine if the poor result was due to deviations or not.

Runtime: On average a complete run of Dromaeo takes 30-40 minutes and this provides results in many different areas of the web browser.

SunSipder

SunSpider is made by Apple and is currently, March 2009, in version 0.9. Apple also maintains Webkit, which is an application framework, and Safari, which is a web browser mainly used on OSX systems. This benchmark suite does contrary to Dromaeo rely solely on slow running tests, meaning that it only runs a test five times to measure the performance. The tests only cover the JS engine. This approach is not a problem as long as the tests are updated continuously else the test will become fast running tests, as described earlier and become error prone.

Runtime: On average a complete run of SunSpider takes 25 seconds.

V8

V8 Benchmark Suite is made by Google and is currently, March 2009, in version 3, v8 also refers to the JS engine Google uses in their web browser Chrome. The tests in V8 cover only the JS engine as with SunSpider but a major difference between V8 and SunSpider is that V8 utilizes the approach, explained earlier in *How*, where it is the time for each test that is fixed as opposed to the number of runs. The whole suite consist of six tests and each of these tests is run for at least one second and based on this a score is calculated. The score is calculated by comparing the benchmark results with a reference system, that have a score of 100.

Runtime: On average a complete run of V8 takes eight seconds.

Final thoughts

Each of these three benchmark suites take on their own approach to benchmarking a web browser and the JS engine, but in our context of client benchmarking on runtime. Non of them can be run fast enough, so that they would not cause reduced performance for the client and be unnoticed. The V8 benchmark suites comes close to being usable for live usage. The concept with limiting the time a test executes instead of the number of runs will be used in the development of the JS benchmark tests in our monitor. This helps us giving guarantees on the runtime of the benchmark and it will lower/remove maintenance of the each test in the benchmark.

2.2 JavaScript

JavaScript was created at Netscape by Brendan Eich in 1995 for their Netscape 2 browser. In 1996 Microsoft developed JScript, which was a dialect of JS that added several features. Later in 1996 Netscape submitted JS to Ecma International, in order to create a standard. This resulted in the ECMA-262, which defined the ECMAScript scripting language. Today both JScript and JS aim to implement the third revision of ECMAScript[18], which is the current standard.

The reason that JS became such a success, is amongst other things, that it is integrated directly in the browser, unlike Java or Flash, which requires a third party plug-in to be installed. Furthermore JS is a dynamic language, which allows a quick trial and error approach to development.This kind of evolutionary/explorative programming makes sense, since JS is a prototype based object oriented language, inspired by Self [19]. Prototypes allows JS to create objects from nothing, meaning there is no need to predefine classes, as known from several mainstream Object Oriented Programming (OOP) languages.

JavaScript is widely used in web development, however it can be applied in other contexts. Rhino[20] is a JS implementation written in Java developed by the Mozilla Foundation, and it can be used to integrate JS in ordinary Java applications. In Mozillas own web browser, Firefox, JS is used to create browser extensions, which allows users to enhance Firefox with new features. Yahoo! Widgets[21] are small applications, which resides on top of the windows desktop, these widgets are also created using JS. We will however not concentrate on this application of JS, but rather focus on JS in web development.

2.2.1 Limitations

Even though JS is widely used, it does have some limitations, which other languages such as Java does not.

The current trend in hardware development, with regards to CPUs, is to have more cores per chip. This calls for parallelization of applications in order to utilize the extra processing power. Unfortunately as it is right now, JS is single threaded, which means it, up to a certain point, will not benefit from the new generation of CPUs. Google have created a browser plug-in that, amongst other things, allow JS to be sent to a worker pool, thus achieving simple threading. This plug-in is however only available for Firefox and Google Chrome.

Another limitation of JS in web browsers is, that it is restricted to a sandbox execution environment in order to secure the end-user. Due to this JS has very limited access to information on the running system. The only system information accessible to JS is through the DOM. The DOM holds information such as which browser the user is running, on which operating system, and whether or not Java is enabled. The sandboxed environment is however an understandable security precaution, but unfortunately it does not provide much system information. If it was possible to access more information about the system, it could have been used in CLIMB, both for the benchmark part and the information given to the user. Information which could be useful to have access to is memory, CPU, hard drive usage, as they could explain poor performance.

As mentioned in section 1.1, there are several different web browsers. These web browsers can be broken up into groups by layout engines, combined with JS engines. The major engines are:

Presto The layout engine of all the Opera web browsers. Presto has its own JS engine, and the current version of this is called Futhark. It is however normally just referred to as the Presto engine.

Trident The layout engine that have been used by Internet Explorer since IE 4. Like Presto, Trident doubles as a JS engine, or more correctly a JScript engine.

Gecko The Mozilla Corporation layout engine used in amongst other things their web browser Firefox. The JS engine used with Gecko is Spidermonkey, although this is to change with Firefox 3.1 which will feature the Tracemonkey engine.

WebKit The layout engine used by Apple Safari and Google Chrome. Safari is just about to change from the JavaScriptCore engine, developed for WebKit, to SquirrelFish Extreme in response to the current market development. Google Chrome launched with their own V8 JS engine.

With this many different JS engines there could be a giant mess of features and incompatibilities. Fortunately however they all comply with the ECMAScript standard reversion 3. That is not to say that each of the engines do not add new and extended functionality to JS. But in order to ensure that web applications work in the different JS engines, one must as a developer refrain from using features not supported by the majority of engines.

It is however not only the JS functionalities that change from engine to engine, the DOM implementation also vary. The DOM is a representation of elements in the current page. Seen from a JS perspective the top level DOM object is `window`. The `window` object holds reference to all elements on a web page, this means that all JS functions are accessible through it. A function defined in an external JS file called `niceLittleFunction` can

be retrieved through the `window` object like so: `window.niceLittleFunction`. However as a general rule of thumb the `window` object can be omitted. The `window` object also provides access to the entire HyperText Markup Language (HTML) document through the `window.document` object. It is also possible to get information on the the browser, and installed plugins. Figure 2.1 shows a simplified representation of what is accessible through the DOM. A cloud represents a swam of objects which depends on the browser used and the specific page being viewed.

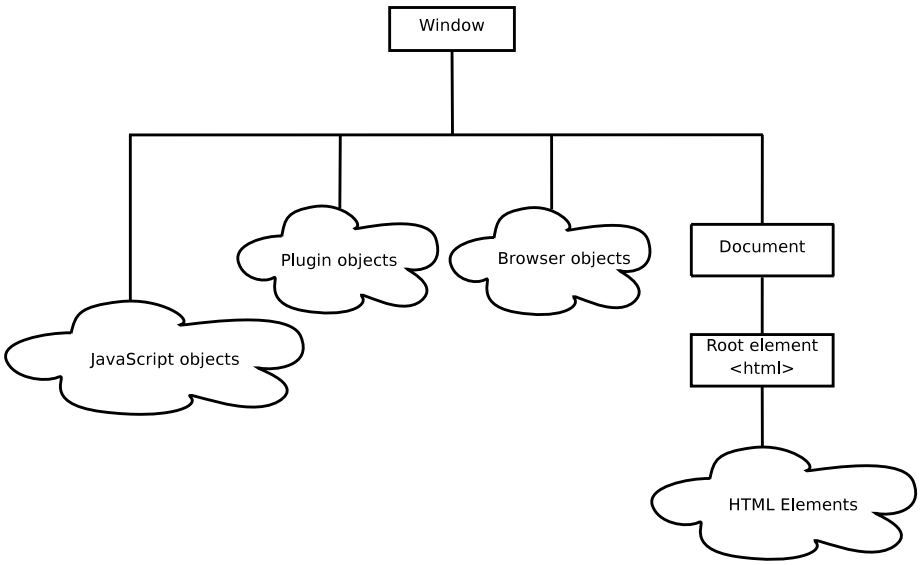


Figure 2.1: A simplified representation of the DOM as seen in JavaScript

In order to ensure that the DOM would not become a mess the World Wide Web Consortium (W3C) started to standardize it. This lead to the release of the DOM Level 1 specification in 1998, and in 2000 the DOM Level 2. The later DOM specifications are however not as widely supported as e.g. the ECMAScript standard.

In order to interact with the content of web pages, JS has access to the DOM. Besides providing access to all HTML elements, the DOM also stores references to all JS on a web page. This means that by traversing the DOM, it is possible to inspect all defined JS objects, including functions. JavaScript can access HTML elements using a series of DOM functions, such as `getElementById` and `getElementsByTagName`. JavaScript elements, unlike the HTML elements, does not reside in a special object. As figure 2.1 shows JS objects resides in the `window` object, alongside with browser specific objects, and plug-in variables like Java and Flash. Furthermore it is possible to access the HTML elements through the `document` object. This means that there is no easy way to obtain a reference to all the user defined JS objects. So in order to locate e.g. all the non-native JS functions on a web page, one would have to inspect the objects in the `window` object. To avoid searching through irrelevant objects like the HTML elements, one could exclude the `document` object from the search. This does however not take care of all the native JS functions, and other irrelevant objects in `window`. It is therefore not a trivial matter to locate user defined functions and objects in the DOM.

2.2.2 Reflection

Reflective programming stems from early Artificial Intelligence (AI) research[22], where self-awareness was desirable. In order to achieve self-awareness, a program must be able to obtain information of its current state and behavior. Ancona and Cazzola mentions that *"Reflection is defined as the activity performed by an agent when doing computations about itself. This activity involves two aspects: introspection (state and structure observation) and intercession (behavior and structure alteration)."*[23]. To put it a bit simpler, in order to be reflective a program has to be able to inspect and evaluate its current state, and modify its behavior, e.g. modify its code. With this notion of reflective programming we now turn back to look at JS.

As mentioned, JS has access to all JS objects on the current page. All JS objects can be explored by doing an exhaustive search and each time an object is found, it is examined for functions. Functions have a native function called `toString()`, which returns the string representation of the current function. This make it possible to inspect user defined functions, it is however not possible to do this with native browser functions as these just return [`native code`]. It is therefore possible to treat functions as mere strings, which can be manipulated and then used to replace the original function. Using these features of JS, it is possible to do reflective programming. An example of this is the jsSerializer[24]. The jsSerializer traverses a given object and outputs it as a string in JS object notation.

Using the reflective properties of JS it is possible to locate user functions and rewrite these.

2.2.3 JavaScript Co-existence

CLIMB is planned to be implemented in JS and the CLIMB code will be included on every page. Therefore it will be mixed with the JS, which the developer created for the web application.

Introducing foreign JS into existing working JS in a web application can give some complications as the foreign code can interfere with the original code. These interferences can be *name clashing* and *blocking/starvation* and they will be treated below.

Name clashing occurs when functions or variables in the same scope share the same name. This will under normal circumstances not happen as the developer would have written all the code and thereby have all names under control. When foreign code is included, name clashing can occur if functions or variables in the foreign code share names with original code. An example of name clashing is shown in code example 2.1 and executing `foo(4);` would result in only one of the functions being executed. Depending on the JS engine it differs, which one is getting executed.

```
1 // Original code from originalCode.js
2 function foo(bar)
3 {
4   var f = bar;
5   return f;
6 }
7
8 // Foreign code from foreignCode.js
9 function foo(bar)
10 {
11   var f = Math.random() * bar;
12   return f;
13 }
```

Code example 2.1: *Name clashing example in JS*

Name clashing can be avoided in two ways, either by wrapping all the foreign code into an object or by having a unique naming convention. Creating an object for all foreign code will create an entire scope for all the foreign code and it can then only be reached through the object and thereby avoiding name clashing. The other solution is to prepend or append some well defined word to all global variables and functions. Both solutions are illustrated in code example 2.2.

```
1  // Original code from originalCode.js
2  function foo(bar)
3  {
4    var f = bar;
5    return f;
6  }
7
8  // Foreign code wrapped in an object
9  function CLIMB()
10 {
11   this.foo = function(bar)
12   {
13     var f = Math.random() * bar;
14     return f;
15   }
16 }
17
18 var climb = new CLIMB();
19
20 // Foreign code prepended with the word climb
21 function climbFoo(bar)
22 {
23   var f = Math.random() * bar;
24   return f;
25 }
```

Code example 2.2: *Name clashing example in JS*

The name clash problem is easy to avoid/fix but another problem can occur when foreign JS code is introduced. Foreign code can block the original code with functions that run for a large amount of time or perhaps for a infinite time period, this is called starvation. Starvation occurs when the foreign code contains some form of a loop, it could simply be an infinite `while`-loop or a series of functions calling each other forming a loop. The loop will block the original code, which is unfortunate because it is probably not the intended effect.

This style of programming is not standard practice in JS as it is event driven, but it is necessary in the monitor context, as it runs throughout the runtime of the application.

An example of this problem is shown in code example 2.3, the foreign code consist of an infinite `while`-loop which blocks the events in the original code.

```
1  // HTML code from index.html
2  <button onclick="fooClick()">Click Me</button>
3
4  // Original code from originalCode.js
5  function fooClick()
6  {
7    alert("You clicked on me!");
8  }
9
10 // Foreign code from foreignCode.js
11 function fooLoop()
12 {
13   var f;
14   while(true)
15   {
16     f = Math.random();
17   }
18 }
19 fooLoop();
```

Code example 2.3: *Infinite loop blocking JS events*

A way to go around this problem would be to utilize threads but unfortunately JS does not support threads natively and therefore they can not be used to solve this problem. Threads can be simulated in JS but would not increase performance. The best possible outcome would be unchanged performance but this is highly unlikely.

Since using threads is not a viable solution to this blocking problem, the only other possible solution available in JS is to use timing events to accomplish co-existence between the foreign and the original JS code. In JS timing events is a way to delay the execution of a function, either once or repeatedly and an example of timing events is shown in code example 2.4. The timing event `setTimeout` delays the execution of a function with a defined time in milliseconds whereas `setInterval` also delays the execution but it does so infinitely. So in code example 2.4 the execution of `fooOnce()` will execute `fooDelay()` after 20 seconds and the execution of `fooRepeat()` will execute `fooDelay()` every 20 seconds until the event is reset.

```
1  // Timing events in JS
2  function fooDelay()
3  {
4    alert("I'm delayed by 20 seconds");
5  }
6
7  function fooOnce()
8  {
9    setTimeout("fooDelay()", 20000);
10 }
11
12 function fooRepeat()
13 {
14   setInterval("fooDelay()", 20000);
15 }
```

Code example 2.4: *Example of timing events*

Timing events makes it possible to simulate infinite loops without total blockage of other events. The infinite loop can be simulated with `setInterval`, which as explained above repeatedly call a function with a certain delay. The delay between each iteration and the fact that it is an event which executes the iteration, which makes it possible to trigger other events, while having an infinite loop. When `setInterval` is executed, it queues an event in the JS event queue and after the specified amount of time, the event will, if idle, be triggered and the desired function will be executed. Events triggered after the timed event will be executed as normal.

```
1  // HTML code from index.html
2  <button onclick="fooClick()">Click Me</button>
3
4  // Original code from originalCode.js
5  function fooClick()
6  {
7    alert("You clicked on me!");
8  }
9
10 // Foreign code from foreignCode.js
11 function fooLoop()
12 {
13   var t;
14   t = setInterval("fooMath()", 1000);
15 }
16
17 function fooMath()
18 {
19   var f;
20   f = Math.random();
21   alert(f);
22 }
23 fooLoop();
```

Code example 2.5: *Example of timing events implementing a while-loop*

This simulation of an infinite loop is of course only appropriate in situations where the interval between each iteration is not critical as the timed event can be delayed more than the requested amount of time. Such situations could be that of a monitor, which would be some form of an infinite loop but where the execution of the monitor comes second to the execution of the original JS code. An example of this is shown in code example 2.5, here the function `fooMath()` will be executed every second and each time the button is clicked the function `fooClick()` is run as opposed to code example 2.3 where all is blocked due to the infinite `while`-loop.

2.3 Existing Web Monitors

Web monitors take many shapes and they have different purposes. Some of the existing monitors have already been introduced in section 1.2 and they will now be analysed.

2.3.1 User Interaction Tracking

Atterer et al. developed a HTTP proxy called UsaProxy, that is capable of injecting JS in existing pages. The proxy is then used to monitor user interaction with any kind of web site and web application. The idea stems from a proxy logger developed by Hong et al. called WebQuilt [25]. WebQuilt however was minded at static web sites. UsaProxy parses each web page and updates all links to route through the proxy, and each link is assigned an unique identifier. When the user presses a link the proxy will log the link pressed and serve the next page to the user. This works well for static web sites, but with RIAs emerging Atterer et al. decided to expand the logging possibilities to include mouse and keyboard tracking. All this is done without interfering with existing JS.

The UsaProxy is implemented such that anything that is not a web page is transparently served to the users web browser. However if it is a web page, the proxy will simply inject a JS file into the head section of the HTML document, and the header of the request is altered to reflect this.

The tracking is done by utilizing the event handlers in JS, using `addEventListener()` (W3C) and `attachEvent()`(IE). This insures that it does not interfere with the original JS application. The UsaProxy client side tracking triggers on events such as `onfocus` for input fields, and `onclick` for links and buttons. However in order to relive the client from logging, every mouse movement and scroll action must be logged but instead of this, Atterer et al. have chosen to use a function which is invoked at regular intervals. This also means that the log does not become bloated with log entries of mouse movement.

The actual logging is done using a JS image object, where the `src` parameter is used to fetch a specific image from the UsaProxy. The url for the image is used to piggyback log data as parameters, and the UsaProxy server is setup to output a sanitized version to the log files. This log data can then late be used to recreate the user interaction by simply replaying the logged events on e.g. a screen shot.

What Atterer et al. have achieved is to create a monitor for tracking user interaction with web sites. It has a reasonably small footprint, and it is fully capable of co-existing with the original JS of the site. As a result of the latter, it can be applied to any site, without having to intrude on either the developer or test user. The monitor is therefore well suited for the purpose it was build.

The question that remains is, what knowledge can gained from the UsaProxy monitor? The way Atterer et al. have chosen to implement their monitor is quite interesting, however

it is mostly applicable for a small scale user group. This is because, if it were to be included for all users, it would put the proxy under heavy load. Therefore it would be smarter to compile the changes that the UsaProxy makes into the files served instead of changing the files on the fly.

The use of periodic invocation, used to capture mouse and scroll movement, is more interesting and in line with the main idea of this report. The same goes for the co-existence, which is achieved using event handlers.

2.3.2 AjaxScope

AjaxScope[10] is a monitor platform implemented as a proxy, using many of the same ideas as UsaProxy. Its purpose is to allow developers to monitor and analyze large web applications when these are used by many users. In order to do this, developers create instrumentation policies, which are used to identify JS code structures, and rewrite these with logging. Amongst the instrumentation policies implemented are: infinite loop detection, general error logging, and memory leaks. Furthermore these instrumentation policies can also be used to conduct drill-down performance profiling, where only potentially slow function calls are further examined. To keep down the footprint of monitoring, AjaxScope does not inject all policies to a client request, instead it shuffles between them. Each client is registered in form of a session id, such as to allow AjaxScope to shuffle the policies per client. Again, like the UsaProxy, all this is done without interfering with existing JS code.

AjaxScope is implemented in C#, and it uses a custom JS parser which is based on ECMAScript version 3[18]. The JS code of a web site is parsed into an Abstract Syntax Tree (AST), which is then passed through each of the instrumentation policies.

The idea of not applying all filters and tests at once, but instead cycling thorough them over time, is quite interesting. It makes sense not to burden the client with a monitor which has a larger load than that of the site which it monitors.

2.3.3 JPU

Andrea Giammarchi has developed a simple little JS library, that adds a CPU activity gauge to a site [26]. It allows the user to visually see a continuously updated approximation of the current CPU activity. Activity is represented by a little bar chart as known from the Microsoft Windows Task Manager.

To achieve this the `setInterval()` function, available in JS, is used, `setInterval()` was described in section 2.2.3. It is set to call a function, which exploits the inaccuracies of `setInterval()` during load. If the client is under heavy load the event created by `setInterval()` will not be executed on time and is therefore delayed for some time, this delay can be used to measure the load of the client. If the function is set to be executed every 500ms and the delay e.g. is 250ms, a total of 750ms between the run of the function, then the CPU is under heavy load.

So JPU is able to show a somewhat crude picture of the current load of the CPU. This approach to measuring CPU activity is adequate for the type of benchmarking in this report and JPU does provide the basis idea for monitoring CPU activity.

Chapter 3

Problem Statement

The vision for this project and related work has been introduced in chapter 1. In chapter 2 both JavaScript benchmarking and JavaScript has been analyzed. The existing monitors, which was first introduced in related work in section 1.2, was also analyzed further to create the base of this problem statement.

Our previous work[2] shows that RIAs are becoming more and more common and it is leading trend within web development. Major web sites have already followed this trend and make heavy use of JS to achieve advanced RIAs. To the modern computers running the latest browsers, such as Mozilla Firefox, Google Chrome, and Apple Safari, this does not pose big problems, since they all have achieved fast JS execution speeds using new and improved JS engines. However these engines will allow developers to deploy even more advanced JS on web sites/applications, and this is bound to have an impact on the user experience for weak clients such as those using, old computers, mobile phones, or even just outdated browsers. This situation confronts the developer with a dilemma. Should he create a RIA, or restrain himself and create a not so rich internet application in order to comply with the weaker clients? Our vision is that a client monitor library written in JS will aid the developer in creating RIA, by giving him a tool which allows him in differentiating the JS code run on strong and weak clients.

This dilemma and our vision leads us to state the following hypotheses, which will clarify if our vision is realizable:

JavaScript Benchmarking

As stated in section 2.1, the JS benchmark suites on the market are not cut for being used in "live" situations, as they take a long time to run and puts the client under a heavy load. This means that if the existing JS benchmark suites were to be used to measure load on a client it would create an unwanted waiting time, and increase the load on the client, which will lower the user experience for weak clients even more.
Therefore it is hypothesized that:

H1: *By creating a series of fast running JavaScript (JS) tests, it will be possible to perform live benchmarking of a client with little performance loss for the client.*

H2: *Using a few fast tests instead of adapting the large scale testing used by the benchmark suites analyzed, will be sufficient to determine the load of the client.*

Existing web monitors

The monitors analyzed in section 2.3 do not fulfill our vision described in section 1.3. Either the monitors are for debugging purpose or used only to monitor certain areas on the client, which is not enough to advance the development of RIA. Therefore it is hypothesized that:

H3: *A library which integrates a monitor, and runs entirely on the client, can be useful for manipulating existing JS code.*

H4: *The user experience can be preserved, when live benchmark results are used to modify the original JS code.*

Over all it is hypothesized that:

H: *A library containing a continuous performance monitor will aid the developer in creating RIAs which are accessible for all kinds of clients, weak or strong.*

In order to confirm these hypotheses, we propose a JS library, which uses a continuous performance monitor. The library should comply with the vision presented in section 1.3. This means that the library has to be able to gather information about the client's status, consisting of current load, and network status. Using the performance monitor, the library should be able to utilize the gathered information and apply it to modify existing JS code. The modification of existing JS code could be handled by something like the instrumentation policies in AjaxScope. This would allow a developer to modularize his modifications, making them easily reusable. The library's monitor should be light enough, so as not to interfere notably with the execution of the existing JS.

The next chapters will describe the work done to confirm the hypotheses, and it starts with the design and implementation of the CLIMB library. This is followed by a test of CLIMB, and thereafter an evaluation.

Chapter 4

Design & Implementation

In this chapter the application description, design and implementation are presented. This is followed by a description of the implementation process, where the different stages of development are explained. The set up in which CLIMB has been developed is then described, along with the different difficulties that arose during development. Lastly the less trivial parts of the implementation, including benchmark testing, filters, and the instrumentation of these, will be described in detail.

4.1 Application Description & Design

The prototype, which we are going to develop in this project, is a JS library which continuously benchmarks the client. It will also be able to modify existing JS code by adjusting it to the current state of the client. The library is called CLIMB, which is short for Continuous Live Information Monitor and Benchmarking, and it was first introduced in chapter 1 and then again in section 1.3.

CLIMB consist of two main components, a benchmark component and a filter component. These two components are what makes the library work and they will be described throughout this section and choices made in the design will be justified. The benchmarking component will, as the name suggest, benchmark the client in different ways ranging from load test to network connectivity. The filter component handles code modification. The modifications are applied based on filters in CLIMB, which can either rewrite entire functions or add additional code before and/or after a function body.

The CLIMB library is meant to be included on web sites along side the other JS libraries and markup. It has been designed so that it does not interfere with the original JS code. The user will not notice the benchmarking of his web browser and will only notice that the functionality of the RIA degrade as his load is high.

The two components will in the following sections be described in more detail. First the benchmark component is described in section 4.1.1, and then the filter component in section 4.1.2.

4.1.1 Benchmarking

CLIMB will benchmark the client using a series of different tests, ranging from client load to network connectivity. The benchmarking component consists of two sub components, a

benchmark framework and the benchmark tests. These will be described individually, as a concerted description would be too chaotic. The individual benchmark tests must naturally be tied to the framework.

V8[17] and Dromaeo[14] have been used as inspiration for the design of our benchmark framework and benchmark tests, since they have taken an interesting approach for benchmarking, which can prove useful for this project. First the benchmark framework is described and then the benchmark tests.

Benchmark framework

CLIMB will, as mentioned, use benchmark tests to identify bottlenecks on the client. In order to make this process run as smoothly as possible, there will be a framework which keep track of all the benchmark tests. The tests will be run several times throughout a visit to the web site, as the client status can change over time. Therefore the status will need to be reevaluated continuously by running the tests again. The reevaluation of the status has to be done several times and the interval between each run has to be chosen such that the client does not suffer noticeable load.

If the benchmark tests are run all the time, all the execution time available will go to these, and thereby block the execution of the important parts, namely the JS which drives the web site. But if the benchmark tests are run to far apart, then the client could have change status several times without it being reflected on the web site. This will not give a adequate representation of the client status and CLIMB will therefore not be able to take the right actions.

So what is the right interval between benchmarks? Through a series of small experiments, done during the development, it was found that CLIMB should run benchmark tests every five seconds. Five seconds gives a reasonable representation of the client status, while test run every two seconds on the other hand was found to interfere during high load. As the the idea with CLIMB is that it should be able to perform live benchmarks there were no reason to test with a greater interval. The five seconds are however just an interval which could be changed if evidence supports it.

The CLIMB benchmark framework will be tightly integrated in CLIMB, and it will therefore look as if it is CLIMB that runs the benchmarks.

The following list gives a feature overview of the framework.

- Add/Remove benchmark tests.
- Run tests continuously.
- Collect results from tests.
- Calculate a total score based on the results.

Implementation wise the framework itself will be a JS function accessed through the monitor, which contains the tests, the results, and the total score for the client. The filter component can access the score and base its actions upon this value.

Benchmark tests

The benchmark tests themselves, will be non-complex tests in order to keep the running time of each run to a minimal. This means that each test should consist of as few components

as possible. A low runtime for a test is important since a test will only be given a certain amount of time to run as many times as possible. If the runtime of a test is too long then the results will be too error prone and unreliable. This is because if a certain amount of time has been allocated for a test to run, and the benchmark operation itself takes too long to run then the test data can become skewed. If the benchmark test has been given 300 milliseconds to execute, and the benchmark operation takes 100 milliseconds to execute, then there is very little room for measuring the current load. In this case the benchmark test would be able to perform 3 operations, as opposed to 150 if the operation took 2 milliseconds. The latter forms a better basis for deciding the load of the client.

A test in CLIMB can be anything and it can be used for any purpose. A test can simply be used to measure the load on the client or measure its string operation performance. Even though a test might be aimed at testing a clients string operation performance, the load of the client can also be derrived as a heavy load will influence the result of the test.

An example of a benchmark test could be simple string operations like concatenating a character to a string as many times possible during the a period of time. This simple test would say something about the load of the browser, but if it were made a bit more advanced and used other string operations it would give some estimate of the string capabilities of that particular client. There are several other things besides from string manipulation, which can be used for tests. such as objects, regular expressions, arrays and all other JS components. The Dromaeo test suite uses all these to benchmark browsers.

To ensure that a test will only run for a specific time period, the test needs to be designed in a manner, which support this. An example of the way used in CLIMB can be seen in code example 4.1, where the limit has been set to 500 milliseconds. The 500 milliseconds are of course too much time to spent on a single benchmark test as the other JS would be unresponsive for half a second. For the implementation a period of 100 milliseconds has been chosen, as this would allow the tests to be run enough times and only put load on the client for one tenth of a second every fifth second. This way of limiting the time of a benchmark test is similar to the one used in V8[17], and it is used as it gives certainty about the maximum runtime of a test.

```
1  function benchmarkTest()
2  {
3    //records the time spent
4    var time = 0;
5    //the start time of the test
6    var startTime = new Date();
7
8    for (var n = 0; time < 500; n++)
9    {
10     //actual test here
11     time = new Date() - startTime;
12   }
13 }
```

Code example 4.1: *A template for benchmark tests.*

The test template in code example 4.1 uses the `Date` object in JS to generate the current time and compare it to the start time on every run of the test. This use of the `Date` object produces an overhead for each run of a test but it is necessary to make sure that a test only runs for a certain amount of time. Since the overhead is added to all tests, the results are all effected equal. In JS the only way to measure time is by using the `Date` object, which represents real time.

4.1.2 Filters

The other component of CLIMB is the filter component, which main task is to handle the filters and the use of these. The filter component is dependent on the benchmark component in order to work as intended. Without the benchmark results, then the filters could not be used to alter existing functions to take load and network into consideration, meaning that CLIMB would not fulfill its purpose. The benchmark component supplies the client status information that will be used by the individual filters to optimize the JS. This will ultimately allow the user to get the best experience possible on his system.

The filter component traverses the DOM to find all user defined functions. A user defined function is all the functions which the developer has created or included. Included libraries such as jQuery, Prototype, etc. also counts as user defined functions. CLIMB uses the filters to decide which functions should be affected by the current client status and what to do with the functions in question.

A filter can rewrite functions and give them new meanings. The rewriting can be done in two ways. Either the rewriting overwrites the whole function body or the AOP concepts `before` and `after` are used. The `before` and `after` blocks can be used to insert code before/after the function body, hence their names. The filter design has been inspired by AjaxScope and it keeps the filters as simple as possible. Before and after have been added as it convenient to be able to add new code to functions without changing the existing code.

A list with examples of possible filters:

Fading A filter which can change all functions that gradually changes an objects opacity, such that they first checks the current load of the client. If the current load is above a certain level, then the functions would just show or hide the object instead of making a fade effect.

Animation Along the same lines as the fading filter, many JS libraries allow developers to animate HTML elements. A filter which during high load disables these animations, could be helpful on weak clients.

AJAX information AJAX has become very popular over the last couple of years. Many web applications use AJAX for server communication[2]. However for AJAX to do this the client must logically enough be connected to the network. Most JS libraries does not tell when the client has lost the network connection. A filter which could disable functionalities that uses AJAX in the web application, when the network connection is lost. Another possibility is to inform the user that he is currently not connected to the Internet.

Sorting In cases where the client is very weak it might be an idea to be able to move operations such as sorting data from the client to the server.

Advanced calculations Like the idea with sorting, a filter could be used to move resource heavy calculations to the server if the client is weak.

Library specific A filter or a set of filters could be created for a specific JS library. Where more advanced strategies could be applied to help weak clients.

Filters can be created by the developer in order to suit the needs he has in specific situation. Another approach as mentioned above is to create filters for popular JS libraries

such as jQuery[4] and Prototype[5] etc. This way could be be sufficient for many developers who rely on these libraries.

A filter will be represented by a JS object and must contain a `matching` function and a `rewrite` function. An example of a filter is presented in code example 4.2. The match function is used to define which function(s) the developer wishes to intercept and change the behavior of. This can be done by either looking in the functions for certain properties, or in the case of library specific filters, by looking for specific function names. What the filter is going to change in the matched functions is defined in the rewrite field. A generic function for modifying functions can be used to rewrite the relevant parts of the function. This generic function will support inserting code before and after a function. It will also make it possible to simply overwrite functions.

```
1 var filterName = {
2   "matching": function(){
3   //code here
4   },
5
6   "rewrite": function(){
7   //code here
8   }
9 };
```

Code example 4.2: *The filter design.*

4.2 Development Process

The development of the monitor has been done in stages, where each stage results in a working prototype addressing concerns like co-existence, which was analyzed in section 2.2.3.

All the stages for this project are listed below and they are more thoroughly described in the following sections. The stages were formed from the application description and design in section 4.1.

The rest of this chapter will follow up on all these stages with details on the implementation and additional functionalities.

Development stages

Stage 1(Co-existence): The main goal of this stage was to create a foundation, which ensures that the web site code and the monitor code can co-exist.

Stage 2(Filters): The goal was to make a working prototype of the monitor, which with the use of filters can change user defined functions.

Stage 3(Benchmark framework): Create a benchmark framework such that it is easy to add benchmark tests to the monitor and manage the benchmarking.

Stage 4(Benchmark tests): Create a some fast running benchmark tests.

Stage 1(Co-existence)

The main goal of this stage was to create a foundation, which ensures that the website code and the monitor code can co-exist. To achieve this goal, a RIA has been emulated with a website containing simple effects created in JS. Then the first part of CLIMB, the monitor, was created. It includes a main monitor which handles the execution of benchmark tests. In order to make the monitor work, a benchmark test was created. The test was inspired by

JPU from section 2.3.3, and it utilizes the inaccuracy of the JS timer used in `setInterval`. It does so by measuring the time actually spent between the execution of `setInterval` and the execution of the function given as input to `setInterval`. The test will later be a part of the monitor and be used to indicate the load of client.

Stage 2(Filter)

The second stage was focused on being able to locate user defined functions in the DOM, and create a simple filter. To locate user defined functions, CLIMB traverses the DOM, and prunes the search as progress is made. Each of the located functions are stored, such that they later can be retrieved and manipulated by filters.

When CLIMB was able to traverse the DOM and locate user defined functions, a simple filter was build from the idea described in the design in section 4.1.2.

The filter that was created is a very basic performance profiler. All functions are instrumented with a start and stop time, which is then logged for later analysis. A function for modifying the other functions was then created from the experience obtained by developing the simple filter. The idea with this function is to make it easier to create filters, since all function modification should be alike. More advanced filters are supposed to be developed along with more benchmark tests in stage four.

Stage 3(Benchmark framework)

Creating a benchmark framework was the main goal of this stage. The main idea behind the framework is to make it easier to manage benchmark tests. A simple scheduler was created to shuffle through benchmark tests, this also allows multiple benchmark tests to be added to CLIMB. A description of the framework can be found in section 4.1.

Stage 4(Benchmark tests)

Create a series of fast running benchmark tests which can be added to the benchmark framework created in stage 3. Thereby providing CLIMB with data about the client performance.

In this stage two benchmark tests were created, and added to the monitor. The benchmark tests individually adds performance measurement, and Internet connectivity testing. The two benchmark tests have been, as mentioned earlier, limited to an execution time period of 100 milliseconds. The Internet connectivity test however runs asynchronously and the restriction will therefore not be needed. The performance test is inspired by Dromaeo as it is ideal for our implementation.

The two test are listed below:

String manipulation: This test will concatenate a character to a string, as many times as possible during the 100 millisecond time period. The string manipulation will test the browser capability to handle strings and they are important in JS, because many things are represented with strings.

Internet Connectivity: This test will use the the `XMLHttpRequest` object to test if the client is connected to the internet or not. This test is very useful for RIAs, which also use the `XMLHttpRequest` object to interact with the server, as it will give the developer the opportunity to provide the client with some kind of error handling when he is offline.

Furthermore in stage 4 two filters were created in order to test the capabilities of CLIMB. First a filter that rewrites functions, which instantiates `XMLHttpRequest` objects, to warn the user if he is not connected to the Internet. The second filter is a jQuery specific filter, which rewrites jQuery to disable animations when the browser load is above a certain level.

4.3 Implementation Concerns

Often during development several problems arises, problems which had not been considered during design phase. The development of CLIMB is no different. There have been several concerns during the development, which will be elaborated in the following.

eval() Common practice in JavaScript development is to avoid the `eval` function for several reasons. What the `eval` function does is, take a string, parse it, and execute it as JS code. This makes it a very powerful function, which can be quite dangerous to use if the string source is not trusted. In the case of CLIMB, `eval` is not supposed to be used on user input, which means that this concern can be dismissed. Generally debugging becomes more challenging [27], as code executed using `eval` does not have any line numbers.

In CLIMB `eval` is used more often than might be good, this is partly due to some complications which arose during development. During the development of the CLIMB explorer, when Google Chrome was used for some debugging, every object that was passed on and traversed using the associate array notation yielded an undefined object. The solution to this problem was to use `eval` instead of the associate array notation. An example of how this works can be seen in code example 4.3.

```
1 /* Example of how eval can be used to obtain a reference to an object */
2 var obj = eval("window.nice_object");
3
4 /* Obtaining the same object using assosiative array notation */
5 var obj = window["nice_object"];
6
7 for(o in obj){
8     /* This yeilded an undefined object in Google Chrome */
9     var newObj = obj[o];
10
11    /* Using eval instead yeilded the expected object */
12    var newObj = eval(object_name +"."+o);
13 }
```

Code example 4.3: An example of how to use eval.

Since `eval` is slow compared to the alternatives[27], it is our belief that CLIMB can benefit from a reduction of the number of `eval` invocation. However `eval` is necessary in the filters for replacing existing functions, as they need to recreate the rewritten functions.

try-catch As with `eval` the `try-catch` statement can be extremely useful. This is especially the case, when the CLIMB explorer uses the `toString` on objects, which is not possible in all browsers. It is however not very effective to use `try-catch`, as is explained by Opera developer Mark Wilton-Jones[28]. The `try-catch` allows CLIMB to prune object early in browsers like Firefox and Google Chrome, while still being able to run in other browsers.

Pruning As there are many more objects accessible via DOM, than those which are interesting to CLIMB. Some objects can therefore be disregarded already when they

are encountered at first. As just described the CLIMB explorer uses the `toString` method to prune non user objects from the search. This means that it will not search needlessly through e.g. HTML, and browser specific elements.

There is however a need to prune some user objects during exploration, this is because object can have circular references. Where one object has a reference to another object, which then has a reference to the first object. To deal with this, the CLIMB explorer adds a boolean value to objects, which have been explored. This boolean is then used to avoid exploring already visited objects. An alternative solution would be to keep a reference to visited objects and then prune using this. This would have to be implemented with a hash table, which is not a part of JS and would therefore have to be created. Therefore the added boolean was used in favor of the hash table.

Security During development security has not been taken into account. This means input to functions are not validated, and generally the different functions of climb are only meant to be run by CLIMB itself.

Analysis The analysis or exploration of the DOM is only done once, more specifically when the whole web page has been loaded. This means that functions added after startup, e.g. with the `eval` function, will not be the subject of the rewrite filters. However due to performance and readability most JS libraries do not modify their behavior or add code after page load, only data is being added to the web site continuous. It is therefore not considered a problem as such.

4.4 Implementation

In the following the different parts of the CLIMB prototype will be explained. This includes, how CLIMB can be integrated in an existing web page, how benchmarking is handled, and how filters are applied.

4.4.1 Implementation Premise

The final prototype has been developed in JS, using a coding style closely resembling the style of JavaScript Object Notation (JSON)[29]. Development was done in an ordinary text editor, while debugging was done using Firefox with FireBug[30], and Google Chromes development terminal. This means that CLIMB runs in both Firefox and Google Chrome. In the upcoming chapter other browsers have been tested as well. The prototype is developed as a self embedding JS library, which means that the only thing that has to be done in order to use it is include the relevant JS files.

4.4.2 Initialization

CLIMB has been developed as a self initializing JS library, which means in order to use it, it need only be included on a web page. This is done by adding three functions to the `window.onload`, which will handle the analysis, rewriting and starting the benchmarks. Measure has been taken to make sure that CLIMB does not interfere with other JS scripts, which would otherwise be added to `window.onload`. A function called `addOnLoad`, which allows multiple functions to be added to the event handler, was created and can be seen in code example 4.4. What it does is simply wrap the old `onload` function in a new function where both are executed.

```
291 /* Generic function to add multiple functions to onload */
292 function addOnLoad(obj,func){
293     var oldOnLoad = obj["onload"];
294     /* Is there already an onload event?*/
295     if(typeof(obj["onload"]) != "function"){
296         obj["onload"] = func;
297     }else{
298         obj["onload"] = function(){
299             if(typeof(oldOnLoad) == "function"){
300                 oldOnLoad();
301             }
302             func();
303         }
304     }
305 }
```

Code example 4.4: The addOnLoad function.

The possibility to create self executing function in JavaScript is used in code example 4.5, where the tree CLIMB functions are added.

```
307 (function(){
308     addOnLoad(window,climb.explore.getAllUserFunctions);
309     addOnLoad(window,climb.filters.run);
310     addOnLoad(window,climb.monitor.run);
311 })();
```

Code example 4.5: The CLIMB initialization.

When the web page has loaded, the analysis of the JS code will be run, then the filters will be applied to the relevant functions, and then the benchmark monitor will be started. The benchmark monitor uses a simple scheduler to run the different benchmarks. This scheduler is explained in the next section.

4.4.3 Scheduling

One of the things, that was desired in CLIMB, was the possibility to run several different benchmarks, and do this continuously. This requires some sort of scheduler, that can handle when to run the different benchmarks. To address this a simple scheduler was implemented.

The scheduler is set to execute at a certain interval, which is set to two seconds at default. When it is invoked, it will remove a benchmark test from its internal queue of benchmarks and execute it. The test will then be put in the back of the queue, this repeats at every invocation thus cycling the list of benchmark tests. How it is implemented can be seen in code example 4.6.

```
189     "monitor" : {
190         "mTime"     : null,
191         "running"   : false,
192         //To be used on the onload event
193         "run"       : function() {
194             if(!this.running){ //Only have one instance running
195                 this.running = true;
196                 this.mTime = setInterval("climb.monitor.nextModule()", 5000);}
197         },
198         "tests"     : ["climb.string_test.start","climb.jpu.start","climb.network.
                start"],
199         "nextModule": function(){
200             var f = this.tests.shift();
201             this.tests.push(f);
202             eval(f+"();");//run the referenced function
203             climb.debug.info("Load: "+climb.current_load);
204         }
205     },
```

Code example 4.6: The scheduler implementation.

There are two points of interest in code example 4.6, the `run` function and the `nextModule` function. The `run` function is the initial function which is supposed to be run on load. When it is called it checks if there is already an instance running, if so it will not start the scheduler. In the case that it is the initial call the scheduler will create a `setInterval` instance which will call `nextModule` function at the specified interval. The `nextModule` function handles the execution of the individual benchmark tests. It cycles through the benchmark tests currently stored in the `tests` array by using it as a First in First out (FIFO) queue.

4.4.4 Measuring Current Load

As can be seen from code example 4.6 in the previous section, the benchmark tests are added to an array in the scheduler. These tests should consist of a variety of performance tests, as discussed in section 4.1. There are two kinds of benchmark tests in the current CLIMB implementation: browser load, and network status. Each load test supplies CLIMB with an estimated value of the current load, while the network status test supplies a boolean. The load value is defined to be an integer value in the range 0-5 inclusive. This was chosen because it provides a balanced overview of the current load, allowing a usable, but yet fuzzy, overview of the current browser load. Furthermore the benchmarks used for monitoring the current load are not accurate enough to give more than a rough estimate.

All benchmark tests must provide at least one function, which executes the test. Each test should not run for more than 100ms, and the result of the test must be converted and assigned to the relevant CLIMB status variable, i.e. current load or network status. In order to give a better understanding of how benchmark tests in CLIMB work, an adaptation of JPU will be described below. JPU was introduced in section 2.3.3.

```
48          "start" : function(){
49              if(!this.running){
50                  this.running = true;
51                  this.D = new Date();
52                 this.cTime = setInterval("climb.jpu.get_load()",this.interval);
53              }else{ climb.debug.info("Skipped running jpu-test");}},
```

Code example 4.7: The start function from the JPU test

The first function to explore is the start function, which is meant to be run by the scheduler. As can be seen from code example 4.7, `jpu.start` first checks if it is currently running in order to avoid multiple instances running. It then saves the current time and sets the `jpu.get_load` function to run with an interval of 500ms.

```
62          "get_load"  : function(){
63              this.D = new Date - this.D;
64              this.history.push(this.D);
65              if(this.itter<this.nb_runs)this.itter++; else this.stop();
66                  this.D = new Date;
67          }
```

Code example 4.8: The JPU load gathering function

When the 500ms has passed `jpu.get_load` will be invoked, at which point the current time is used to obtain the time elapsed since the invocation of `jpu.start`. If the browser is under load the time elapsed before the invocation of `jpu.get_load` is greater than 500ms. The time span is then pushed into the history array, to be used later for calculating the mean delay. If the predefined number of invocation has been reached the `jpu.stop` function is called. Lastly the current time is then stored, and if the `jpu.stop` function was not called, then after 500ms `jpu.get_load` will be invoked again. This can all be seen in code example 4.8.

```
54          "stop"  : function(){
55              clearTimeout(this.cTime);
56              this.mean = (this.history.sum()/this.history.length);
57              this.meanPush(this.mean);
58              this.history = [];
59              this.itter = 0;
60              this.to_load();
61              this.running = false;},
```

Code example 4.9: The JPU stop function

When `jpu.get_load` has been run a certain number of times, the `jpu.stop` function will be called. It is responsible for ending the execution of the JPU test, and updating the CLIMB load status. The first thing `jpu.stop` does is to stop timed execution of `jpu.get_load` by clearing the relevant timer. This can be seen in code example 4.9. The mean delay is then calculated and stored in an array, which over time discards old results. In order to make the JPU test ready for later execution various variables are reset. Before ending execution of the test, the function responsible for converting test results to a CLIMB load value is run.

```
34          "to_load" : function(){
35              var i = this.meanHist.length,
36              load = 0;
37              if(this.meanHist[i-1] > 750)
38                  load++;
39              if(this.meanHist[i-1] > 700)
40                  load++;
41              if(this.meanHist[i-1] > 650)
42                  load++;
43              if(this.meanHist[i-1] > 600)
44                  load++;
45              if(this.meanHist[i-1] > 550)
46                  load++;
47              climb.current_load = load;},
```

Code example 4.10: The JPU load calculation

The inaccuracy of timed events in JS can be used to estimate the current browser load. Code example 4.10 shows how this inaccuracy can be can be used to indicate the current load. Since the CLIMB load value is set to be within the range 0-5, the load can be calculated by checking how many milliseconds the `setInterval` was skewed. A stepping of 50ms is a fair approximation of the current load, this is based on the original JPU script. This concludes how load calculation is done in the JPU benchmark, fairly simple, but yet quite effective.

4.4.5 JavaScript analysis

Having shown how CLIMB is integrated into a web page, and how benchmarking is conducted, it is now time to look at the inner workings of the analysis. In CLIMB this is called exploring, since it searches through the native browser object `window`, looking for non native functions. This is done to allow a set of filters, or policies, to rewrite these functions. The actual rewriting will be covered later in this chapter, but first the `explore` object will be dissected.

Looking for non native functions in JS, is in CLIMB done by recursively traversing the `window` object. As mentioned this is done when the web page has been loaded, which means that all external scripts has been loaded and all the actual functions from these are present in the browser. This is handled by the integration function described in section 4.4.2, which is also responsible for calling `getAllUserFunctions`. What `getAllUserFunctions` does is, it simply calls `getAllUserFunctionsIn` with the argument `window`.

```
217         "getAllUserFunctionsIn" : function(objStr) {
218             if(objStr.match(/\.[^a-zA-Z]$/))
219                 ts = true;
220             else
221                 try{ts = (eval(objStr+".climbVisited") == true);}catch(e){ts=true;}
222             if(ts){
223                 return [];
224             }else{
225                 var mdt = this.getUserFunctionsIn(objStr),
226                 func = mdt;
227                 if(mdt.length > 0 && mdt[0] !=null)
228                     for(var i=0;i< mdt.length;i++){
229                         func = func.concat(this.getAllUserFunctionsIn(mdt[i]));
230                     }
231               return func;
232             }
233         },
```

Code example 4.11: The first part of the CLIMB explorer.

The first of two functions used to find user defined functions is `getAllUserFunctionsIn`, which is shown in code example 4.11. It starts by doing some sanity checking, since Firefox can not use `eval` to retrieve objects that are represented by a single non alphabetic character. An example, taken from an analysis of jQuery is: `jQuery.selectors.<`. So if the `objStr` current objects name consists of a single character it must be alphabetic. Also if `objStr` refers to an undefined object then there is no reason to explore it. As a part of pruning objects, all objects that have already been explored, will not be explored again. When it has been been decided that the current object should be explored, the `getUserFunctionsIn` function is called. It will scan the current object for user defined functions, explore objects recursively, and return the found functions. These functions then need to be explored, since in JS functions can contain other functions. This is done by calling `getAllUserFunctionsIn` with each of the found functions. The result from traversing these functions are joined with the results from the initial pass, and returned.

```
239         "getUserFunctionsIn" : function(objStr) {
240               var funs = [],
241               c_type="";
242               try{obj = eval(objStr);}catch(e){obj = false;}
243               if(!obj || obj["climbVisited"])
244                     return [];
245               for (o in obj){
246                 climb.explore.explored++;
247                 chObj = obj[""+o];
248                 if(chObj === undefined){
249                     try{chObj = eval(objStr+"."+o);}catch(e){ chObj = undefined;}
250                 }
251                 try {c_type = typeof(chObj);} catch(e) {c_type = "undefined";}
252                 if(c_type === typeof(function(){})){
253                     if(!this.isNative(chObj)){
254                         funs.push((objStr==="window"?"":objStr+".")+o);
255                     }
256                 }else if(c_type === "object" && o != "prototype" && chObj !== null){
257                     /*IE & Opera Fix, since they do not allow toString to be called on
                            some objects*/
258                     try{t=chObj.toString().match("object Object");}catch(e){t=true;}
259                     if(t){
260                         funTmp = this.getUserFunctionsIn((objStr==="window"?"":objStr+
                                ".")+o);
261                         if(funTmp.length >0 && funTmp[0] != null)
262                             funs = funs.concat(funTmp);
263                         }
264                     }
265               }
266               if(obj !== undefined)
267                 try{obj["climbVisited"] = true;}catch(e){}
268             return funs;
269           }
270         },
```

Code example 4.12: The second part of the CLIMB explorer.

The `getUserFunctionsIn` function is responsible for identifying the non browser native functions, and further exploration of user defined objects. All this can be seen in code example 4.12. It does this by iterating through the supplied object, detecting the type of each object. Just like in code example 4.11, it starts by pruning objects that have already been explored. Then it iterates through the object, checking if the current member is either a function or an object. To access the object members `getUserFunctionsIn` first tries to use the associative array notation, and if this fails then it tries to access it using the JS `eval` function. As mentioned in section 4.3, the reason for using the `eval` function is that Google Chrome started to return `undefined` objects, when using objects as associative array. If the member object is a function, it then checks it using the `isNative` function, which determines if the function is a native browser function as described in section 2.2.2. Now if it is not a native function, it will be saved in an array using a string, which represents the path to the object. When the member object is an object, it will be searched for user functions by recursively calling `getUserFunctionsIn`. The object is however further explored, if the member object is: the special object `prototype`, `null`, or if it is not a standard object. Along the way the results are collected and merged with previous findings. When the object has been fully explored it is marked as visited, using a boolean value, and the result is returned.

4.4.6 Filters

Filters are used to change the behavior of JS functions throughout a web page. They consists of two main parts: matching, and rewriting, as described in section 4.1. A simple filter could be, to rewrite all functions that use AJAX calls to display an error message when the network connection is lost. This could be done by matching all functions which uses the `XMLHttpRequest` variable, which is used to make AJAX calls, and then wrap these functions in a check on the CLIMB `network_status`. The described AJAX filter can be seen in code example 4.13.

```
92      "matches": function (funStr){
93          var fun = this.getObjFromStr(funStr);
94          if(this.isInsane(fun) || funStr.match(/^climb/))
95              return false;
96
97          return fun.toString().match("new XMLHttpRequest");
98      },
99      "rewrite" : function (funStr){
100         var before = "if(climb.network_status){",
101         after = "}else{AJAXNetwork.warn();}";
102         after +="setTimeout(AJAXNetwork.hide,4500);";
103         climb.debug.info("Rewriting: "+funStr);
104       this.modifyFunction(funStr,{before: before, after: after});
105     },
```

Code example 4.13: Network connection filter

The match function has some additional sanity checks. This is done to ensure that the functions which are matched are, amongst other things, not the match function itself. The rewrite function creates a before and after block which is inserted into the original function using `modifyFunction`. It can add code before and after a function, thereby encapsulating it, or it can simply overwrite the function. Overwriting an existing function is however probably not a good idea, since it can easily break a library. This is done by creating a new function as a string, and then evaluation this string. The details can be seen in code example 4.14.

```
117     "modifyFunction" : function (funStr,mod) {
118       var funStart = /^function.*?\{/,
119         fun = this.getObjFromStr(funStr),
120         oldFun= fun.toString(),
121         newFun="";
122
123         oldFun = oldFun.replace(funStart,"");
124         oldFun = oldFun.replace(/\}$/,"");
125         newFun = fun.toString().match(funStart);
126         if(mod.before) newFun += mod.before;
127         if(mod.overwrite) newFun +=mod.overwrite;
128         else newFun += oldFun;
129         if(mod.after) newFun += mod.after;
130         newFun += "}";
131         eval(funStr+"="+newFun+";");
132   }
```

Code example 4.14: The modifyFunction used to alter functions.

There are a number of functions, such as the `getObjFromStr`, `modifyFunction`, and `isInsane`, that are common for filters. These functions could be bundled together in a CLIMB Filter prototype, which each filter could then extend it with their own specialized functions.

4.5 Summary

Throughout this chapter the application design has been presented, introducing both a benchmark framework and the notion of filters. A filter is used to identify specific functions, and modify these, while a benchmark is used to gather information about the client browser. Following the application design was the development process, where the different stages of CLIMB development was described. The development process fell in four stages, first the foundation of CLIMB was created. This includes the basic structure of the monitor with the ability to run benchmark tests. A single benchmark test, based on Andrea Giammarchi's JPU[26], was developed, and integrated into CLIMB. The second stage was to traverse the DOM to locate user defined functions, which are needed in order to apply CLIMBs filters. Stage three was to extend and improve the monitor, setting up a benchmark framework which can continuously run multiple benchmarks. The last stage, stage four, was to create more benchmark tests, which were integrated into CLIMB.

A short specification was then presented, along with several implementation concerns. The concerns were elaborated and addressed the performance of CLIMB mostly, such as the usage of `eval`, and pruning of the DOM search. Finally the details of the CLIMB development were described in detail. As a last note for the development, the full source of CLIMB can be read in appendix A.1, and the implemented filters can be fund in appendix A.2.

Chapter 5

Test

This chapter describes the tests designed to test CLIMB, and at the same time evaluate the hypotheses stated in chapter 3. First the premises of the test will be described, to give the background. Then browser compliance is tested, to see if CLIMB works with other browsers than those used during development. Thereafter CLIMB is applied to a range of different JS libraries, to see how it handles these. Lastly two filters are tested to show that CLIMB can be used for the purposes stated in the hypothesis.

5.1 Test Premise

All the tests will been performed on a laptop with a 2GHz Core2Duo processor, running both Ubuntu 9.04 and Windows Vista SP1. The browsers, used to test CLIMB in, are Firefox 3.0.10, Chromium 3.0.183.0 (Open Source chrome), Opera 9, and Internet Explore 8. The last, IE8 was the only browser that was run under Windows. Throughout the remainder of this chapter Firefox will be used as the reference browser.

Further more all the tests will be set up in a developer controlled environment, where CLIMB will be tested with different JS libraries, to see how it handles these. Two filters will be used to test if the benchmark tests works as intended, i.e. the network detection and the current load tests.

5.2 Browser Compliance

In order to examine whether the CLIMB prototype is able to run in different browsers, a test, in which CLIMB is run in the browsers listed in the previous section, was created. The idea of the test is to firstly see if CLIMB works, but also to see the performance difference amongst the browsers. If the implementation of CLIMB works on said browsers, then it is able to run along side most of the existing cross browser JS libraries.

A simple test page has been created to test browser compliance, where the only scripts included are the two files relevant to CLIMB. Debug information was inserted into the `explore` object to time the analysis in each browser. In order to give a fair run time the test was run ten times in each browser, which was then used to find an average execution time. The initialization function described in section 4.4.2 was used to test for browser compatibility. When the web page has been fully loaded it will initiate the DOM analysis to locate user defined functions.

Name	Status	Functions found	Objects	Analysis time (ms)
Firefox 3	✓	49	232	331
Chromium 3	✓	52	449	143
Opera 9	✓	49	166	(774)
Internet Explore 8	×	N/A	N/A	N/A

Table 5.1: *Test results of different browsers running climb.*

The outcome of the test is presented in table 5.1, where it can be seen that CLIMB works in Firefox and Chromium. In the case of Opera, CLIMB does work, but the analysis time varies greatly. Just after startup a pass can take as little as 300ms, as the test page is reloaded it gradually rise to about 5000ms, which is a bit weird. It might be a flaw in the Presto engine, where page reloads does not clear the JS objects, which could explain why the analysis time seem to increase about 300ms for each subsequent run. It might also be the use of both `try-catch` and `eval` which, as described in the implementation concerns in section 4.3, are not very effective. This inefficiency might also have some kind of deeper impact on the Presto engine, besides being slow. Internet Explore was, unlike the other browsers, not able to run CLIMB. A closer examination of CLIMB using the Visual Studio Web Developer debugger yielded no significant insight into the problem. What was found was that traversing some of the native objects failed in IE. Further testing proved that exploration of the CLIMB object was no problem. The problem therefore lies in the handling and identifying some of the native objects.

Overall the overhead for traversing the DOM is acceptable, we do however believe that it can be done more efficiently. But why is it necessary to make the traversal more efficient if it is already acceptable? Since this test has been conducted on a 2GHz Core2Duo machine, the analysis time is suspected to increase on weaker machines. If the CLIMB explorer is optimized then the analysis time on weak clients would become lower, thus lowering the load time.

Table 5.1 shows that each browser examines a different number of objects. The number of objects varies because it depends on what the browser makes available through the `window` object. Using this we can determine which browser is the fastest when working with CLIMB. This is done by looking at how many functions per millisecond each browser is able to process, and the overall execution time. Chromium is the fastest in terms of both execution time and traversed objects per millisecond, which makes it the fastest. In the test Chromium traversed 3 objects per millisecond. Firefox is the second fastest browser, it traverses 0.7 object per millisecond, making it four times slower than Chromium. The slowest browser in this test was Opera, which in average processed one object in 4.7 millisecond.

When CLIMB is added to a page with no other JS it adds 49 functions, each of the compatible browsers found all of these. One might notice is that Chromium locates three functions more than the other browsers. These are however functions which are native to Chromium, but not marked as such. The functions in question are `external.AddSearchProvider`, `chromium.Interval`, and `chromium.GetLoadTimes`.

Name	Status	Functions found	Objects	Analysis time (ms)
jQuery	✓	354	958	1756
Dojo Toolkit	✓	259	843	1628
Ext JS	✓	1337	1820	3265
Script.aculo.us	✓	5272	6260	15461
MooTools	×	N/A	N/A	N/A
Yahoo! UI Library	×	N/A (239)	N/A (666)	N/A (1145)

Table 5.2: *The result of adding CLIMB to a static site with a JS library included in Firefox.*

5.3 Integration

The purpose of this test is to see if CLIMB is able to co-exist with different JS libraries. During development it has been run along side with the JS library jQuery, which helped to sort out some bugs. jQuery was chosen since is both simple to use, light weight, and widely adapted. There are however a great deal of other JS libraries, which was not used during development. In order to cover these in a sensible way the following will be examined for each JS library:

- Did it work?
- The number of functions found.
- Time used on analyzing, averaged over ten runs.

Since we are not able to test every JS library in existence, we will focus on those that have visual effects, and are commonly used. The tests will include the following JS libraries: jQuery, Yahoo! UI Library, Script.aculo.us, MooTools, Dojo Toolkit, and Ext JS. These libraries falls into two categories, which are standalone libraries, and JS frameworks. The first category focuses mostly on providing JS functions that can amongst other things animate, make AJAX calls, make specific element selection, or manipulate data. JavaScript frameworks, like Yahoo! UI Library and Ext JS, on the other hand also provides developers with GUI elements which can be used to enrich web applications. The frameworks are included in the test, since they allow the same functionality as the ordinary libraries.

Each library is included into a CLIMB test site, which allows for debug logging. The only library that has gotten some special treatment is the Yahoo! UI Library, since it is very modularized. Therefore only the JS files needed to make animations were included. The information gathered from the tests are summarized in table 5.2.

The first thing one might notice is that CLIMB was not able to run together with MooTools and Yahoo! UI Library. With MooTools CLIMB went into an infinite loop, a quick analysis yielded no reason to why. We do however believe that it must be caused by some kind of bug in the pruning process. The same was not the case with Yahoo! UI Library, here they had defined a `toString` function in their animation library, which when explored made CLIMB fail. This was because an object which was referenced in the function did not exist, so when CLIMB tried to explore it, it failed. The problem might be solvable using a `try-catch` block when iterating through an object. We see it as a bug within the Yahoo! UI Library, and believe that it would also make it fail. This has however not been

confirmed. When the `toString` function was out commented, CLIMB reported just fine. The values reported in parenthesis are from the run with the function disabled.

As can be seen from table 5.2, it takes 4, 28ms in average to locate a function in the DOM. This average does not include the CLIMB only test done in section 5.2, as it only shows the general overhead of traversing the DOM and not any specific library. The average do however depend on the object to function ratio in the JS library. To illustrate this a test was set up, where two arrays, each containing 800 JSON objects, were added to the execution of the CLIMB only test from section 5.2. In this test the average execution of the analysis went from 331 to 743, which is nearly a factor of two and a half. This means that the CLIMB analysis will be rather slow when more JS objects are added to the DOM.

The way CLIMB explorer is currently implemented, makes excessive use of the `eval` function, as mentioned in section 4.3. Therefore CLIMB has a performance loss, which should be curable, if the `eval` usage is cut down. With regard to whether or not this has an impact on the user experience, we can only conclude on the basis of the test machine, where the current processing overhead is acceptable, given lighter JS libraries. A study on tolerable waiting time on the web from 2004[31], indicates that users will generally lose interest of a web site, if it takes over 10 seconds to load. It also indicates that a load time of less than two seconds is not registered by the user.

5.4 Filters

As mentioned in section 4.4 filters make up the second part of CLIMB, they should therefore also be tested. We have developed two filters for CLIMB, a general network notification filter that informs the user of network loss in the event of AJAX calls, and a jQuery specific animation filter that uses the current browser load to determine whether or not to use animations. The network filter was explained in section 4.4, but in short it locates functions which uses the `XMLHttpRequest` object. It then adds a conditional section to the functions that, when CLIMBs network status is false, informs the user that the computer is currently not connected to the Internet.

The animation filter for jQuery utilizes the fact that jQuery has an option to toggle effects using the `jQuery.fx.off` variable. Injecting a conditional into the jQuery animation function, that checks the CLIMB `current_load`, will make the animations available depending on the browser load.

Both filters have been created to show how CLIMB can be utilized, i.e. both generally and in a more specific way. The load information could also have been used to turn off e.g. sorting of large HTML tables, and the network status could also be used to make graceful degradation for network communication. It is also possible to use CLIMB with no filters to e.g. disable external links when the network connection is down, this is however not covered but could be an idea for future work.

To test the filters a test page has been created. It uses jQuery to animate a menu which gently fades in when hovered, and gently fades out when exited. Furthermore it has a large table consisting of 2800 cells, which can be toggled to fade in and out. It also has a simple AJAX function from each of the JS libraries, which fetches an array from the web server. Since there are two filters the test will be executed in two steps.

The first test is of the AJAX filter, first testing if the AJAX function can retrieve the data from the web server. Then the network cable is unplugged from the test machine, leaving it without connection to the Internet. The AJAX function is then executed again once CLIMB has detected the connection loss, which may take a little time since the network detection

in CLIMB is not instantaneous. It should then report that there currently is no connection to the Internet. Finally the network cable is plugged in again to see if the modified function behaves properly.

After some minor tweaking, which manly consisted of making CLIMB ask for the right URL, the test ran without problems when testing with jQuery and Dojo Toolkit. Due to the construction of Ext JS, the filter did not work when applied to the Ext JS example. This is because Ext JS uses private function to instantiate the `XMLHttpRequest` object, which CLIMB is not able to inspect. The Yahoo! UI Library was not tested, as it has no small AJAX examples.

The other test concerning the jQuery animations was set up as described and tested by putting the test machine under heavy load. The filter should, when the load rises, disable the animation effects. When the load then returns to normal then the animations should also return. Load during the test was provided by `stress`[32], which is a small utility that can be used to stress test computers. It was run with the following switches: `stress -c 8 -d 3 -i 2`, which means that the CPU will be under heavy load, while the disk is also kept busy. The test page was loaded and `stress` was started, after about ten seconds the effects on the drop down menus stopped fading and was instead shown immediately. When stress was stopped the fading began again, however it was not immediate, this time it only took about two seconds before the change in load was detected.

Both filter tests was concluded in success, performing the rewrite tasks which they were designed for. A minor problem with both filters emerged when the included JS libraries were in compressed form, since a minified JS file makes heavy use of the closure properties of JS. This means that JS functions has access to the variables present when they are created, even after these have disappeared. When a JS file is minified it is compressed and wrapped in a self executing function, where a set of variables hold reference to relevant functions and objects. Therefore when a CLIMB filter attempts to evaluate a rewritten function, then the objects referred to does not exist. Invocation of these functions will then result in an error telling that some single letter object do not exist. This is because when JS is minified all variables are shortened to a single letter.

5.5 Summary

In order to verify the work done in chapter 4 a series of test were setup. All tests were done in a controlled environment. The first test was to see if CLIMB is compliant with other browsers than Mozilla Firefox and Google Chrome. Therefore it was tested using both Internet Explore 8 and Opera 9. In Opera 9 everything ran after some minor adjustments, which consisted primarily of adding a `try-catch` statement. There was however larger problems running CLIMB in Internet Explore 8, which failed while exploring some native objects.

The next set of tests was concerned with CLIMBs ability to co-exist with different JS libraries, and to gather performance data. Six different libraries were tested, of which four worked without any intervention. The Yahoo! UI Library had a conflicting function which could not be examined, when this function was removed the test ran fine. There were one library, MooTools, which went into an infinite loop. Overall the performance was found to be acceptable for lighter JS libraries.

The final set of tests involved the CLIMB filters `AJAXNetwork` and `FadeFilter`. The two filters were tested using an online site, since one of the functions which had to be tested involved AJAX calls. Both tests were successful and showed that the different filter

operations worked, and that the monitor is able to gather information about the client continuously.

Although the functionality of CLIMB has been tested, it has only been in a controlled environment. This is mainly because the current CLIMB prototype showed to have some quirks. A thorough test with a set of existing RIAs would make it possible to answer the **H4** hypothesis stated in chapter 3.

Chapter 6

Evaluation

After the development and testing of CLIMB, comes the evaluation of the work done in those two steps. The first section evaluates on the hypothesis stated in the problem statement chapter 3. Each hypothesis will be discussed and will then be evaluated to see if they hold true. The shortcomings of the development, and the report are then discussed. Shortcomings of the report address the choices made throughout the report. The development shortcomings discuss how usable the final CLIMB prototype actually is. Lastly a set of different features and improvements will be discussed in relation to future work

6.1 Hypotheses

The hypotheses, first stated in the problem statement chapter 3, have so far not been confirmed directly. This section takes up this task and uses the knowledge gained through the implementation and test of CLIMB to confirm or dismiss the five hypotheses. First the four hypotheses **H1** - **H4** will be handled and then lastly the overall hypothesis **H** will be evaluated.

H1: *By creating a series of fast running JavaScript (JS) tests, it will be possible to perform live benchmarking of a client with little performance loss for the client.*
- During the testing of CLIMB, there has not been any noticeable performance loss on the client. The benchmark tests, which were run continuously throughout the lifetime of the test site, did not prevent execution of any existing JS code.

The hypothesis **H1** is therefore found to be *true*.

H2: *Using a few fast tests instead of adapting the large scale testing used by the benchmark suites analyzed, will be sufficient to determine the load of the client.*
- The two performance tests, made during the development, gave an adequate picture of the client status, while being used for testing. When the client was put under stress, it was reflected in the CLIMB load status. Despite the five second delay between each benchmark test run, changes were reflected quickly. The tests did only test overall JS performance of the client.

The hypothesis **H2** is therefore found to be *true*.

H3: *A library which integrates a monitor, and runs entirely on the client, can be useful for manipulating existing JS code.*
- CLIMB has implemented a number of filters, which illustrates how to use the information gathered from the benchmark tests can be utilized. The network status information can be used to inform users of lost connection, and the current load can be used to turn off effects on a site, which use jQuery. Both are examples of situations where a monitor can provide useful information to be used in existing JS code.

The hypothesis **H3** is therefore found to be *true*.

H4: *The user experience can be preserved, when live benchmark results are used to modify the original JS code.*
- If CLIMB is used, the user experience will not become worse as opposed to not using it. This mean that it can be used to remove unnecessary computations when the client is under load, which helps ease the task of running the web site. However in the current state, CLIMB can only help the client in specific cases. Due to the five second delay between the benchmark tests, changes in the client are not reflected immediately.

The hypothesis **H4** is therefore found to be *partially true*.

H: *A library containing a continuous performance monitor will aid the developer in creating RIAs which are accessible for all kinds of clients, weak or strong.*
- The first three hypothesis have been found true, and the fourth is partially true. This supports the idea that a continuous performance monitor is useful when working with RIAs. However to fully confirm this hypothesis, CLIMB would have to support all JS libraries and browsers. Furthermore an extensive RIA would have to be created or take an existing one, where CLIMB is used and tested. More filters would also have to be developed. If this were to happen then we presume that a continuous performance monitor like CLIMB would be an aid to developers.

We therefore conclude that the steps taken in this project have shown that **H** is *plausible*.

6.2 Shortcomings

During the project period choices have been made and when looking back at these choices, some of these could have been taken differently. CLIMB also has some shortcomings, as first stated in chapter 4 in connection with the development itself. This section will also point out shortcomings in the report.

6.2.1 Analysis

During the analysis we looked at JS benchmarking to get ideas for the development of CLIMB. Three major benchmark tools were examined, but we only tested them on one machine. An interesting analysis, which could have helped to classify different clients, could have been to test a wide array of machines, each on a different power scale. The results from this test could have been used in CLIMB and heighten the quality of the classification

done by CLIMB when the benchmark tests are run. Another thing which we could have done was an intense study of how to classify different browsers on different computers.

6.2.2 Development

In chapter 5 it was concluded that CLIMB was able to complete the designed tests, but how usable is the CLIMB prototype really? We know from the performance data gathered that the exploration of the DOM processes about 1 object per two millisecond, which is not very good performance. According to Fiona Fui-Hoon Nah's study of existing literature on the subject[31], there is a diversity in how long users are willing to wait for page load. The general rule of thumb is that anything below 2 seconds is acceptable, and users generally looses interest after 10 seconds load time[31]. With these assumptions CLIMB has an acceptable performance for compact libraries such as jQuery, Dojo Toolkit, where the total analysis time lies around 1.5-2 seconds, which is within the acceptable range of user wait time[31]. With the Ext JS library however the execution time is about 3 seconds, which is, when added to the normal load time of a web page, beginning to border what is acceptable. When larger frameworks, such as Prototype combined with Script.aculo.us, are used, then the overhead of the analysis becomes too high.

Despite the performance of the analysis which is only done at page load, the rest of CLIMB works as intended. The filters are able to rewrite specific functions, as has been shown in section 5.4. It could however have been interesting to have created more filters. Another thing to mention is the incompatibility with MooTools, which was discovered during the compatibility test. This raises the question of what is special about MooTools, such that it ends in an infinite loop, and whether there are other JS libraries, which CLIMB does not work with. However we chose not to look further into, what it was that caused the infinite loop. It might have been interesting to see what caused it, we however suspect that it most likely is some flux in the process of pruning objects.

Lastly it could have been interesting to see how CLIMB would fare on older machines and on mobile devices. Since CLIMB had some problems running under Internet Explore we chose not to expand the tests to mobile devices.

6.2.3 Test

The tests conducted in chapter 5 were all done in a developer controlled environment. As there were some complications with CLIMB, such as not working with MooTools, it was decided not to test CLIMB on an advanced RIA. Therefore to properly test CLIMB a full scale test should be done, where CLIMB is deployed with different RIAs.

CLIMB is supposed to allow weak clients to run heavy RIAs, it has however not been tested on any weak clients. As this is one of the main ideas behind CLIMB, it would only be natural to conduct a series of tests on several weak clients.

The vision behind CLIMB also states that it should help to improve the user experience, therefore a user test, done in an usability lab, would make sense. This way a concrete impression of what CLIMB is capable of could be formed. These usability tests should reflect the intended use of CLIMB, which means that they should be used on both weak and strong clients.

6.3 Future Work

CLIMB is as described far from done and this section will describe possible changes and additional features. The changes and additional features are things we have discovered during the development process and they will be described below:

Avoid `eval()`: Avoid the use of `eval()` when traversing the DOM and use the associative array style. This should boost the load of CLIMB significantly, as using associative array style of accessing functions is faster than using `eval()` on object strings.

Extensive client testing: Do a thorough benchmark testing of a selected range of clients, ranging from weak to strong clients. The results would be used to improve the classification of clients.

Optimize pruning: When CLIMB traverses the DOM, it marks all objects with a boolean it visits to ensure that an object is only visited once. To optimize this, a hash table with object references could be used instead.

Filters Create more built-in filters such that all JS libraries are supported. Make sure that all aspects are covered, such that all demanding effects used in RIAs can be catched by the filters per default.

Filter - Advance the rewrite functionality: Give the developer more power when using the filters. As it is now, it is only possible to rewrite the whole function or add additional code before and/or after the function body.

Filter - Advance the matching functionality: Examine if it is possible to match on other than just functions and expand it to objects and variables. An example of this could be to match just the `XMLHttpRequest` object instead of the function, which uses it, and e.g. change the url if the main server is down.

User testing The ultimate test of CLIMB would be to perform a user test, where real users are asked to test RIAs, which have CLIMB included, on both strong and weak clients. This test will give a final verdict on CLIMB, as it will show if it preserves the user experience on weak clients.

Chapter 7

Conclusion

As we showed in our previous work *Unified Multitier Web Development*[2], web development anno 2008/09 is about RIAs. This often involves multiple languages, e.g. HTML, CSS, ActionScript, and JS. It was concluded that multiple languages can be an unnecessary burden on the developers, especially newcomers, and that a unification of the languages aids the developers, thereby allowing them to create better RIAs.

So unifying RIA development, eases the development, this leads to more RIAs which are more advanced/demanding application. This situation, combined with the progress that has happened with JS-engines, is good news for the developer.

However with the release of the Apple Iphone, the number of mobile clients have exploded and the developers of RIAs are now faced with a new problem: weak clients. Weak clients are, unlike a modern computer with a modern browser, not fit to run heavy web applications.

This project deals with differentiating the JS run on the client, such that the user experience is maintained despite of the client type weak or strong. Five hypotheses have been stated in chapter 3. They revolve around two subjects, benchmarking and self modifying code, and what happens when they are combined. The hypotheses has been named **H1** to **H4**, while the main hypothesis **H**. **H1** and **H2** are concerned with benchmarking the client, while **H3** and **H4** are about whether this information can be used to purposely alter user defined functions. The main hypothesis **H** combines all four hypotheses into the following:

> **H:** *A library containing a continuous performance monitor will aid the developer in creating RIAs which are accessible for all kinds of clients, weak or strong.*

The hypotheses were stated based on the information presented in both the introduction in chapter 1, and the analysis performed in chapter 2.

In chapter 1 the trends on the "web market" and the work related to ours were presented. This formed the basis for an analysis of JS benchmarking, which dealt with the three leading browser benchmarking frameworks, Dromaeo, SunSpider and V8. JavaScript was then analyzed in order to get a deeper insight prior to the development phase. Finally the existing web monitors, which were first shortly introduced in the introduction, were analyzed further to gain a better insight in how they work.

Following the analysis was the problem statement in chapter 3, which was based on the knowledge gained in the preceding two chapters. The problem statement introduced the five hypotheses, which covers the whole project. In order to confirm these hypotheses, we proposed a JS library, which would benchmark the client and modify the JS code to reflect

the outcome of the running benchmark tests. This library is called CLIMB, which is short for Continuous Live Information Monitor and Benchmarking.

The design and implementation of CLIMB was described in chapter 4. It provided insight into the decisions made prior to the development of CLIMB, and it presents the interesting aspects of the development. In order to answer the hypotheses the developed prototype needed to be tested, therefore CLIMB was tested in chapter 5. The tests were designed to address the hypotheses and they covered the following areas, browser compliance, integration, and filters. First of CLIMB was tested in different browser to see to which extent it worked. It was able to run in Mozilla Firefox 3, Chromium 3, and Opera 9, but it did not work in Internet Explorer 8. Afterwards CLIMB was run with six different JS libraries in Firefox. Of the six libraries two caused problems with CLIMB, these were MooTools and Yahoo! UI Library.

Lastly the two filters, which were created during the development, were tested. The first filter was a network filter, which uses the network status information gathered by the CLIMB monitor. It matches functions that instantiates the `XMLHttpRequest` object. These are then rewritten, such that AJAX calls, which are made when the client is offline, will tell the user that the network connection is currently down. The other filter was created to disable graphical effects in jQuery when the client experiences high load. Both filters worked as expected and the test was concluded a success.

The test of CLIMB formed the basis for the evaluation of this project, and the hypotheses stated in the problem statement. Evaluation of the hypotheses was done by looking at the test results, and by using the experience gained during the development. The hypotheses **H1**, **H2** and **H3** were all found to be *true*, while **H4** was found to be *partially true*, as the tests done in chapter 5 did not provide enough evidence to fully support it. The main hypothesis **H** was found to be *plausible*, as there is still work to be done in order to confirm it fully. It is however our belief that if CLIMB is tuned and put through a series of real life test, then the hypothesis **H** will be found to be true. The evaluation of the hypotheses can be found in section 6.1.

Some of the knowledge gained through the project have come too late, for it to be reflected properly in the project. This knowledge have been used to pinpoint the shortcomings of the project. Section 6.2 describes these shortcomings and form the basis for the section about possible future work for CLIMB. Since CLIMB is still just a prototype, and the hypothesis **H** was not fully confirmed, there is still room for future work. This was discussed in section 6.3, where it was clarified what should be done to finalize CLIMB, and thereby confirm hypothesis **H**.

In our last report we showed that RIAs are evolving and the demands for the client are raising[2]. We have tried to solve the problem of running RIAs on weak clients by creating a JS library for benchmarking and client adaption. As the tests in chapter 5 indicates, it is possible to make existing JS code adapt to changes in the clients status. Although the vision of CLIMB is that is should be able to allow RIAs to be run on weak clients, it will not be able to eliminate the need to optimize JS code during development. CLIMB needs more work in order to work with real life RIAs, but we have taken a good step towards a solution for the weak clients.

Bibliography

[1] Kevin Mullet. The essence of effective rich internet applications, 2003. http://www.gevasys.de/materialien/essence_of_ria.pdf.

[2] Daniel S. Korsgård, Morten Bøgh, Michael S. Knudsen, and Markus Krogh. Unified multitier web development, 2008. Dat5 report at Aalborg University.

[3] Adobe. Adobe flash cs4 professional, 2009. http://adobe.com/products/flash/.

[4] Jquery.com. jquery: The write less, do more, javascript library, 2009. http://www.jquery.com/.

[5] Prototypejs.org. Prototype javascript framework: Easy ajax and dom manipulation for dynamic web applications, 2009. http://www.prototypejs.org/.

[6] Google. Official gwt web site, 2008. http://code.google.com/webtoolkit/, Visited 05/12/08.

[7] Inc NextApp. Echo web framework, 2008. http://echo.nextapp.com/site/, Visited 05/12/08.

[8] Microsoft. Measuring browser performance: Understanding issues in benchmarking and performance analysis. 2009.

[9] Richard Atterer, Monika Wnuk, and Albrecht Schmidt. Knowing the user's every move: user activity tracking for website usability evaluation and implicit interaction. In Les Carr, David De Roure, Arun Iyengar, Carole A. Goble, and Michael Dahlin, editors, *WWW*, pages 203–212. ACM, 2006.

[10] Emre Kiciman and V. Benjamin Livshits. Ajaxscope: a platform for remotely monitoring the client-side behavior of web 2.0 applications. In Thomas C. Bressoud and M. Frans Kaashoek, editors, *SOSP*, pages 17–30. ACM, 2007.

[11] Phu H. Phung, David Sands, and Andrey Chudnov. Lightweight self-protecting javascript. In R. Safavi-Naini and V. Varadharajan, editors, *ACM Symposium on Information, Computer and Communications Security (ASIACCS 2009)*, Sydney, Australia, March 2009. ACM Press.

[12] Microsoft. The official microsoft silverlight site., 2008. http://silverlight.net/, Visited 05/12/08.

[13] John Resig. John resig - javascript programmer, 2009. http://www.ejohn.org/.

[14] John Resig Mozilla. Dromaeo: Javascript performance testing, 2009. http://www.dromaeo.com/.

[15] Mozilla Foundation. Mozilla.org - home of the mozilla project, 2009. http://www.mozilla.org/.

[16] Apple Webkit. Sunspider javascript benchmark, 2009. http://www.webkit.org/perf/sunspider-0.9/sunspider.html.

[17] Google V8 Team. V8 benchmark suite, 2009. http://v8.googlecode.com/svn/data/benchmarks/v4/run.html.

[18] E. C. M. A. International. *ECMA-262: ECMAScript Language Specification*. ECMA (European Association for Standardizing Information and Communication Systems), Geneva, Switzerland, third edition, December 1999.

[19] David Ungar and Randall B. Smith. Self: The power of simplicity. *Lisp and Symbolic Computation*, 4(3):187–205, 1991.

[20] Mozilla Foundation. Rhino - javascript for java, 2009. http://www.mozilla.org/rhino/.

[21] Yahoo! Inc. Yahoo! widgets: useful, fun, beautiful little apps for mac and windows, 2009. http://widgets.yahoo.com/.

[22] J.M. Sobel and Daniel P. Friedman. An introduction to reflection-oriented programming, 1996. http://www.cs.indiana.edu/~jsobel/rop.html.

[23] Massimo Ancona and Walter Cazzola. Implementing the essence of reflection: a reflective run-time environment. In Hisham Haddad, Andrea Omicini, Roger L. Wainwright, and Lorie M. Liebrock, editors, *SAC*, pages 1503–1507. ACM, 2004.

[24] Iconico. The workshop - javascript serializer allows you to serialize to xml and javascript strings, 2009. http://www.iconico.com/workshop/jsSerializer/.

[25] Jason I. Hong, Jeffrey Heer, Sarah Waterson, and James A. Landay. Webquilt: A proxy-based approach to remote web usability testing. *ACM Trans. Inf. Syst.*, 19(3):263–285, 2001.

[26] Andrea Giammarchi. Jpu - javascript cpu monitor, 2007. http://webreflection.blogspot.com/2007/09/jpu-javascript-cpu-monitor.html.

[27] Eric Lippert. Fabulous adventures in coding: Eval is evil, part one, 2003. http://blogs.msdn.com/ericlippert/archive/2003/11/01/53329.aspx.

[28] Mark 'Tarquin' Wilton-Jones. Efficient javascript, 2006. http://dev.opera.com/articles/view/efficient-javascript/.

[29] Json.org. The json homepage, 1996. http://www.json.org/.

[30] Inc. Parakey. Firebug - web development evolved, 2008. http://getfirebug.com/.

[31] Fiona Fui-Hoon Nah. A study on tolerable waiting time: how long are web users willing to wait? *Behaviour & IT*, 23(3):153–163, 2004.

[32] Amos Waterland. The stress project page, 2008.
http://weather.ou.edu/~apw/projects/stress/.

Appendix A

Appendix

A.1 CLIMB source code

```
1  /**
2   * Add a sum function to arrays.
3   *
4   */
5  Array.prototype.sum = function(){
6      for(var i=0,sum=0;i<this.length;sum+=this[i++]){}
7      return sum;
8  };
9
10 /*
11  *  The main climb object
12  *
13  */
14 var climb = {
15     /** Current load lies between 0=idle and 5=heavy load */
16     "current_load" : 0,
17     "jpu" : {
18         "cTime"     : null,
19         "history"   : [],
20         "mean"      : 0,
21         "meanHist"  : [],
22         "nb_runs"   : 4,
23         "running"   : false,
24         "itter"     : 0,
25         "interval"  : 500,
26         "D"         : null,
27         "meanPush"  : function(mean){
28             if(this.meanHist.length <= 3){
29                 this.meanHist.push(mean);
30             }else {
31                 this.meanHist.shift();
32                 this.meanHist.push(mean);
33         }},
34         "to_load" : function(){
35             var i = this.meanHist.length,
36             load = 0;
37             if(this.meanHist[i-1] > 750)
38                 load++;
39             if(this.meanHist[i-1] > 700)
40                 load++;
41             if(this.meanHist[i-1] > 650)
42                 load++;
43             if(this.meanHist[i-1] > 600)
44                 load++;
45             if(this.meanHist[i-1] > 550)
46                 load++;
47             climb.current_load = load;},
48         "start" : function(){
49             if(!this.running){
50                 this.running = true;
51                 this.D = new Date();
52                this.cTime = setInterval("climb.jpu.get_load()",this.interval);
```

```
53              }else{ climb.debug.info("Skipped running jpu-test");}},
54          "stop"  : function(){
55              clearTimeout(this.cTime);
56              this.mean = (this.history.sum()/this.history.length);
57              this.meanPush(this.mean);
58              this.history = [];
59              this.itter = 0;
60              this.to_load();
61              this.running = false;},
62          "get_load"  : function(){
63              this.D = new Date - this.D;
64              this.history.push(this.D);
65              if(this.itter<this.nb_runs)this.itter++; else this.stop();
66                  this.D = new Date;
67          }
68      },
69      "string_test" : {
70          "mini": 0,
71          "maxi": 0,
72          "max_mean" : [],
73          "running": false,
74          "strConcat": function (){
75              var str="";
76              for(var i=0;i<100;i++){
77                  str += "c";
78              }
79              return str;
80          },
81          "start": function(){
82              if(climb.string_test.running){
83                  climb.debug.info("Skipped running string test");
84              }else{
85                  climb.string_test.running = true;
86                  var elapsed = 0,
87                  usec = 0,
88                  start = new Date();
89
90                  for (var n = 0; elapsed < 100; n++){
91                      this.strConcat();
92                      elapsed = new Date() - start;
93                  }
94                  usec = n/elapsed;
95                  this.to_load(usec);
96                  climb.string_test.running = false;
97              }
98          },
99          "to_load": function(usec){
100             var load = 0,
101             diff=0;
102             if(this.maxi < usec){
103                 this.maxi = usec;
104                 this.append_result(usec);
105             }
106             diff = (this.max_mean.sum()/this.max_mean.length) - usec;
107             if(diff >3)
108                 load++;
109             if(diff >5)
110                 load++;
111             if(diff >7)
112                 load++;
113             if(diff >10)
114                 load++;
115             if(diff >12)
116                 load++;
117             climb.current_load = load;},
118         "append_result" : function(usec){
119             if(this.max_mean.length <= 3){
120                 this.max_mean.push(usec);
121             }else {
122                 this.max_mean.shift();
123                 this.max_mean.push(usec);
124             }
125
126         }
127     },
128     "network_status" : true,
129     "network" : {
130         "xmlhttp": null,
131         "running": false,
132         "start" : function(){
133             if(!climb.network.running){
```

```
134                 climb.network.running = true;
135                 this.checkConnection();
136                 climb.network.running = false;
137             }else{
138                 climb.debug.info("Skipped running network-test");
139             }
140         },
141         "createXhttp" : function(){
142             if (window.XMLHttpRequest)
143             {
144                 climb.network.xmlhttp=new XMLHttpRequest();
145             }
146             else if (window.ActiveXObject)
147             {
148                 climb.network.xmlhttp=new ActiveXObject("Microsoft.XMLHTTP");
149             }
150         },
151         "checkConnection" : function(){
152             if(climb.network.xmlhttp == null)
153                 this.createXhttp();
154 
155             if(climb.network.xmlhttp != null)
156             {
157                 try{
158                     climb.network.xmlhttp.onreadystatechange=climb.network.stateChange
                            ;
159                     climb.network.xmlhttp.open("GET",window.location.pathname,true);
160                     climb.network.xmlhttp.send(null);
161                 }catch(e){
162                     climb.network_status = false;
163                 }
164             }
165         },
166         "stateChange": function(){
167             if (climb.network.xmlhttp.readyState==4)
168             {
169                 try{
170                     if (climb.network.xmlhttp.status==200)
171                     {
172                         climb.network_status = true;
173                         return;
174                     }
175                     else
176                     {
177                         climb.network_status = false;
178                         return;
179                     }
180                 }
181                 catch(err)
182                 {
183                     climb.network_status = false;
184                     return;
185                 }
186             }
187         }
188     },
189     "monitor" : {
190         "mTime"     : null,
191         "running"   : false,
192         //To be used on the onload event
193         "run"       : function() {
194             if(!this.running){ //Only have one instance running
195                 this.running = true;
196                 this.mTime = setInterval("climb.monitor.nextModule()", 5000);}
197         },
198         "tests"     : ["climb.string_test.start","climb.jpu.start","climb.network.
                start"],
199         "nextModule": function(){
200             var f = this.tests.shift();
201             this.tests.push(f);
202             eval(f+"();");//run the referenced function
203             climb.debug.info("Load: "+climb.current_load);
204         }
205     },
206     "explore" : {
207         "uFunc" : null,
208         "explored": 0,
209         "getAllUserFunctions" : function(){
210             if(climb.explore.uFunc == null){
211                 var T = new Date();
212                 climb.explore.uFunc = climb.explore.getAllUserFunctionsIn('window');
```

```
213                 climb.debug.info("All functions retrived, found: "+climb.explore.uFunc
                        .length+" Execution took: "+(new Date()-T)+" Traversed: "+climb.
                        explore.explored);
214             }
215         },
216         /** Returns all userdefined functions in a given object */
217         "getAllUserFunctionsIn" : function(objStr) {
218             if(objStr.match(/\.[^a-zA-Z]$/))
219                 ts = true;
220             else
221                 try{ts = (eval(objStr+".climbVisited") == true);}catch(e){ts=true;}
222             if(ts){
223                 return [];
224             }else{
225                 var mdt = this.getUserFunctionsIn(objStr),
226                 func = mdt;
227                 if(mdt.length > 0 && mdt[0] !=null)
228                     for(var i=0;i< mdt.length;i++){
229                         func = func.concat(this.getAllUserFunctionsIn(mdt[i]));
230                     }
231               return func;
232             }
233         },
234         /** Returns true if Function contains [native code], indicating a non user
                defined function */
235         "isNative" : function(fun){
236             var nativeFunction = /\[native code\]/;
237             return fun.toString().match(nativeFunction);},
238         /** Finds immidiate user defined functions in supplied object */
239         "getUserFunctionsIn" : function(objStr) {
240               var funs = [],
241               c_type="";
242               try{obj = eval(objStr);}catch(e){obj = false;}
243               if(!obj || obj["climbVisited"])
244                     return [];
245               for (o in obj){
246                 climb.explore.explored++;
247                 chObj = obj[""+o];
248                 if(chObj === undefined){
249                     try{chObj = eval(objStr+"."+o);}catch(e){ chObj = undefined;}
250                 }
251                 try {c_type = typeof(chObj);} catch(e) {c_type = "undefined";}
252                 if(c_type === typeof(function(){})){
253                     if(!this.isNative(chObj)){
254                         funs.push((objStr==="window"?"":objStr+".")+o);
255                     }
256                 }else if(c_type === "object" && o != "prototype" && chObj !== null){
257                     /*IE & Opera Fix, since they do not allow toString to be called on
                            some objects*/
258                     try{t=chObj.toString().match("object Object");}catch(e){t=true;}
259                     if(t){
260                         funTmp = this.getUserFunctionsIn((objStr==="window"?"":objStr+
                                ".")+o);
261                         if(funTmp.length >0 && funTmp[0] != null)
262                            funs = funs.concat(funTmp);
263                         }
264                     }
265               }
266               if(obj !== undefined)
267                 try{obj["climbVisited"] = true;}catch(e){}
268             return funs;
269           }
270         },
271
272
273     "filters" : {
274         "list" : [AJAXNetwork,FadeFilter],
275         "run"  : function(){
276             var i=0,
277             j=0;
278             if(climb.explore.uFunc && climb.filters.list.length > 0){
279                 for(i; i < climb.explore.uFunc.length; i++){
280                     for(j=0;j< climb.filters.list.length;j++) {
281                         if(climb.filters.list[j]["matches"](climb.explore.uFunc[i])){
282                            climb.filters.list[j]["rewrite"](climb.explore.uFunc[i]);
283                         }
284                     }
285                 }
286             }
287         }
288     }
```

```
289 };
290
291 /* Generic function to add multiple functions to onload */
292 function addOnLoad(obj,func){
293     var oldOnLoad = obj["onload"];
294     /* Is there already an onload event?*/
295     if(typeof(obj["onload"]) != "function"){
296         obj["onload"] = func;
297     }else{
298         obj["onload"] = function(){
299             if(typeof(oldOnLoad) == "function"){
300                 oldOnLoad();
301             }
302             func();
303         }
304     }
305 }
306 /**CLIMB Initialization */
307 (function(){
308     addOnLoad(window,climb.explore.getAllUserFunctions);
309     addOnLoad(window,climb.filters.run);
310     addOnLoad(window,climb.monitor.run);
311 })();
```

Code example A.1: CLIMB Source

A.2 CLIMB Filters Source Code

```
1  /***************************
2      Filter Skeleton
3  ***************************/
4  var ClimbFilter = {
5      "getObjFromStr" : function(funStr){
6          var obj=false;
7          try{obj = eval(funStr);}catch(e){}
8          return obj;
9      },
10     "isInsane" : function (fun){
11         if(!fun || fun === this.matches || typeof(fun) != "function")
12             return true;
13         else
14             return false;
15     },
16     "matches" : function (funStr){},
17     "rewrite" : function (funStr){},
18     "modifyFunction" : function (funStr,mod) {
19       var funStart = /^function.*?\{/,
20         fun = this.getObjFromStr(funStr),
21         oldFun= fun.toString(),
22         newFun="";
23
24         oldFun = oldFun.replace(funStart,"");
25         oldFun = oldFun.replace(/\}$/,"");
26         newFun = fun.toString().match(funStart);
27         if(mod.before) newFun += mod.before;
28         if(mod.overwrite) newFun +=mod.overwrite;
29         else newFun += oldFun;
30         if(mod.after) newFun += mod.after;
31         newFun += "}";
32         eval(funStr+"="+newFun+";");
33   }
34 };
35
36 var performanceInfo = {
37     "getObjFromStr" : function(funStr){
38         var obj=false;
39         try{obj = eval(funStr);}catch(e){}
40         return obj;
41     },
42     "isInsane" : function (fun){
43         if(!fun || fun === this.matches || typeof(fun) != "function")
44             return true;
45         else
46             return false;
47     },
```

```
48      "matches" : function (funStr){
49          var fun = this.getObjFromStr(funStr);
50
51          if(this.isInsane(fun) || funStr.match(/^climb/))
52          return false;
53
54          return true;
55      },
56      "rewrite" : function (funStr){
57          var bf = "var climb_pf = new Date;",
58      af = "climb_pf = new Date - climb_pf; climb.debug.info(\"Execution took: \"+
            climb_pf);";
59
60        this.modifyFunction(funStr,{before: bf,after: af});
61      },
62      "modifyFunction" : function (funStr,mod) {
63        var funStart = /^function.*?\{/,
64          fun = this.getObjFromStr(funStr),
65          oldFun= fun.toString(),
66          newFun="";
67
68          oldFun = oldFun.replace(funStart,"");
69          oldFun = oldFun.replace(/\}$/,"");
70          newFun = fun.toString().match(funStart);
71          if(mod.before) newFun += mod.before;
72          if(mod.overwrite) newFun +=mod.overwrite;
73          else newFun += oldFun;
74          if(mod.after) newFun += mod.after;
75          newFun += "}";
76          eval(funStr+"="+newFun+";");
77    }
78  };
79
80  var AJAXNetwork ={
81      "getObjFromStr" : function(funStr){
82          var obj=false;
83          try{obj = eval(funStr);}catch(e){}
84          return obj;
85      },
86      "isInsane" : function (fun){
87          if(!fun || fun === this.matches || typeof(fun) != "function")
88              return true;
89          else
90              return false;
91      },
92      "matches": function (funStr){
93          var fun = this.getObjFromStr(funStr);
94          if(this.isInsane(fun) || funStr.match(/^climb/))
95              return false;
96
97          return fun.toString().match("new XMLHttpRequest");
98      },
99      "rewrite" : function (funStr){
100         var before = "if(climb.network_status){",
101         after = "}else{AJAXNetwork.warn();}";
102         after +="setTimeout(AJAXNetwork.hide,4500);";
103         climb.debug.info("Rewriting: "+funStr);
104       this.modifyFunction(funStr,{before: before, after: after});
105     },
106     "hide" : function (){
107         document.getElementById("climb-network-status").parentNode.removeChild(
              document.getElementById("climb-network-status"));
108     },
109     "warn": function(){
110         var div = document.createElement('div'),
111         text = document.createTextNode("You are currently not connected to the
              internet...");
112         div.appendChild(text);
113         div.setAttribute("style","position: fixed;bottom:10px;right:25px;background-
              color:#C00;display:inline;");
114         div.setAttribute("id", "climb-network-status");
115         document.body.appendChild(div);
116     },
117     "modifyFunction" : function (funStr,mod) {
118       var funStart = /^function.*?\{/,
119         fun = this.getObjFromStr(funStr),
120         oldFun= fun.toString(),
121         newFun="";
122
123         oldFun = oldFun.replace(funStart,"");
124         oldFun = oldFun.replace(/\}$/,"");
```

```
125         newFun = fun.toString().match(funStart);
126         if(mod.before) newFun += mod.before;
127         if(mod.overwrite) newFun +=mod.overwrite;
128         else newFun += oldFun;
129         if(mod.after) newFun += mod.after;
130         newFun += "}";
131         eval(funStr+"="+newFun+";");
132   }
133 };
134
135 var FadeFilter = {
136     "getObjFromStr" : function(funStr){
137         var obj=false;
138         try{obj = eval(funStr);}catch(e){}
139         return obj;
140     },
141     "isInsane" : function (fun){
142         if(!fun || fun === this.matches || typeof(fun) != "function")
143             return true;
144         else
145             return false;
146     },
147     "matches" : function (funStr){
148         var fun = this.getObjFromStr(funStr);
149
150         if(this.isInsane(fun))
151             return false;
152         return funStr.match("animate");
153     },
154     "rewrite" : function (funStr){
155         climb.debug.info("Rewriting: "+funStr);
156         var before = "if(climb.current_load < 3){jQuery.fx.off=false;}else{jQuery.fx.
              off=true;}";
157         this.modifyFunction(funStr,{before:before});
158     },
159     "modifyFunction" : function (funStr,mod) {
160       var funStart = /^function.*?\{/,
161         fun = this.getObjFromStr(funStr),
162         oldFun= fun.toString(),
163         newFun="";
164
165         oldFun = oldFun.replace(funStart,"");
166         oldFun = oldFun.replace(/\}$/,"");
167         newFun = fun.toString().match(funStart);
168         if(mod.before) newFun += mod.before;
169         if(mod.overwrite) newFun +=mod.overwrite;
170         else newFun += oldFun;
171         if(mod.after) newFun += mod.after;
172         newFun += "}";
173         eval(funStr+"="+newFun+";");
174   }
175 };
```

Code example A.2: CLIMB Filter Source

A.3 Project Summary

This master's thesis in programming technology is focused on web technologies, more specific Rich Internet Application (RIA) development. The purpose of this thesis is to examine and solve the problems, which RIA developers are facing when creating RIAs targeted at both weak and strong clients.

The thesis starts with an introduction to the new trends in the world of web development, where subjects, such as RIA development, web browsers, and client types, are discussed. The situation is clear, recently web browsers have become faster at executing JavaScript (JS). At the same time the number of weak clients are on the rise, this is because new mobile phones, like the Apple iPhone, are able to browse the web. This situation is bad news for developers, as they have to choose between strong and weak clients, a dilemma that is unwanted.

A solution for this problem is presented in form of a JavaScript library, which continuously performs live benchmarking of the client, and at the same time able to adapt existing JavaScript code by using the benchmarking results.

Prior to the creation of the solution, the basics of web browser benchmarking has been analyzed. JavaScript has also been analyzed to gain better insight prior to creating the library. This information is used later in the design and implementation stage, where a prototype of the JavaScript library is created. As an inspiration for the benchmarking and monitor part of the library, the existing solutions have been analyzed. This includes three different browser benchmarking tools, a debugging and performance profiler for JS called AjaxScope, and a load monitor called JPU.

The problem statement presents five hypotheses which forms the basis for the development of the JavaScript library, which is named Continuous Live Information Monitor and Benchmarking (CLIMB). The first four hypotheses cover JavaScript benchmarking and the ideas on how to utilize the results, while the last hypothesis summarizes the four hypotheses to:

H: *A library containing a continuous performance monitor will aid the developer in creating RIAs which are accessible for all kinds of clients, weak or strong.*

In order to confirm these hypotheses a prototype of the CLIMB library has been created. The development was conducted as an iterative process, executed in four stages, where each stage resulted in a working prototype. CLIMB is a JavaScript library which consists of two parts. The first part continuously benchmarks the client throughout the runtime of the web site. The other part can analyze and modify existing JavaScript code, and change its behavior to reflect the current status. This is done using CLIMB filters, which can match and rewrite the existing functions. During the development of CLIMB, two usable filters were created. One for intercepting all calls to the `XMLHttpRequest` object, and rewrite these to inform the user if the network connection is lost. The other filter can intercept the graphical effects added by jQuery and turn these of, if the client is under heavy load. In order to provide these two filters with information on both client load and network status, some benchmark tests which are targeted at network connectivity and client load were developed.

After the development CLIMB was tested to evaluate if the five hypotheses holds true. The tests performed included testing the compatibility with popular web browsers on the market, the co-existence with six other JavaScript libraries such as jQuery, and the two filters. All the tests yielded good results with only a few exceptions. CLIMB is in its

current state not able to run under Internet Explorer, and one of the tested JavaScript libraries sends CLIMB into an infinite loop.

The evaluation of CLIMB resulted in the confirmation of the majority of the hypotheses and hypothesis **H** was found to be plausible.

In order to fully confirm **H**, further work has to be done on CLIMB. This includes making CLIMB cross browser compatible, testing on real life Rich Internet Applications, testing on multiple weak and strong clients.