Ludwig-Maximilians-Universität München
Institut für Informatik
Lehrstuhl für Programmier- und Modellierungssprachen

Bachelor-Thesis

JSMapReduce - Distributed Computing with Web Clients and LAMP

Philipp Langhans

Supervisor: **Prof. Dr. François Bry**
Advisor: **Christoph Wieser**

Date: June 27, 2012

Erklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig angefertigt, alle Zitate als solche kenntlich gemacht sowie alle benutzten Quellen und Hilfsmittel angegeben habe.

Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

München, June 27, 2012

Philipp Langhans

Zusammenfassung

Durch den technologischen Fortschritt im Bereich von Datenträgern, wachsende Möglichkeiten in der automatisierten Datenerhebung und neue Algorithmen zur Informationsgewinnung und -verknüpfung nehmen die Datenmenge und damit auch die Anforderungen an eine effiziente Bearbeitung solcher Daten in vielen Bereichen der Wirtschaft und Wissenschaft zu. Diese Arbeit beschäftigt sich mit den Möglichkeiten Probleme im sogenannten 'BigData' Sektor durch eine massiv parallele Berechnung mit verteilten Webclients zu lösen. Dazu werden Berechnungen, die sonst auf einem Webserver stattfinden würden, auf verbundene Seitenbesucher ausgelagert. Das Schema der Berechnungen basiert auf MapReduce, wodurch datenparallel in verschiedenen Phasen berechnet wird und die Ergebnisse serverseitig aggregiert werden.

Die gesamte Logik, die für clientseitige Berechnungen nötig ist, lässt sich mit einem einzigen Skript realisieren, welches nebenläufig zu einem Seitenbesuch oder einer Webapplikation in einem modernen Webbrowser ausgeführt werden kann. Durch den verfolgten Ansatz wird idealerweise eine Skalierung der verfügbaren Ressourcen mit jedem zusätzlichen Client, also Verbraucher, bewirkt und so das Ungleichgewicht zwischen Leistungserbringer und -verbraucher reguliert.

In dieser Arbeit wird ein einsatzfähiges Framework zur Verteilung, Aggregation und Kontrolle von Webclient-Berechnungen aufgebaut. Zusätzlich wird dieses Framework mit einem Testsystem verbunden, sodass Testteilnehmer ein Browserspiel spielen können, während sie im Hintergrund ablaufgesteuerte Berechnungen durchführen. Bei den Tests werden umfangreiche Analysen zur Performance, Stabilität und Skalierbarkeit des erstellten Systems erhoben, um Aussagen über mögliche Einsatzgebiete und die Effizienz der Berechnungen treffen zu können.

Abstract

Due to the technological progress in the development of storage media, increasing capabilities in the automated data collection and new algorithms for information retrieval and correlation recognition, both the amount of data as well as the complexity of analyzing these vast data sets are growing in various disciplines such as business intelligence or scientific fields. This thesis discusses the possibilities of solving problems in the so-called 'big data' sector through a massively parallel computation with web clients. For this purpose, calculations that would otherwise take place on a web server, are swapped to and calculated by ordinary site visitors in their browsers. The model behind the calculations is based on MapReduce, resulting in a data-parallel processing in different phases, where partial results are aggregated on the server.

The entire logic needed for client-side calculations can be implemented in a single script which can be executed by most modern web browsers, concurrently to one's web page visit. This approach, ideally, ensures a scaling of available resources with each additional client and thereby regulates the imbalance between service provider and consumer.

In this work, an operational framework is implemented for the distribution, aggregation, and control of such calculations. In addition, this framework is connected to a test system, so that participants can play a browser game, while performing calculations in the background. Extensive analyzes regarding the performance, stability and scalability are made within these tests to evaluate the system and to assess possible application fields.

Contents

List of Abbreviations

AJAX	Asynchronous JavaScript and XML
API	Application Programming Interface
CSS	Cascading Style Sheets
DBMS	database management system
DHTML	Dynamic HTML
DOM	Document Object Model
EDGE	Enhanced Data Rates for GSM Evolution
FCFS	First-come, first-served
GFS	Google File System
GPGPU	General Purpose Computation on Graphics Processing Unit
GPU	Graphics Processing Unit
GSM	Global System for Mobile Communications
GUI	Graphical User Interface
HDFS	Hadoop Distributed File System
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
ISP	Internet Service Provider
MPM	Multi-Processing Module
OS	Operating System
PHP	PHP: Hypertext Preprocessor
RDBMS	Relational Database Management System
RIA	Rich Internet Application
RPC	Remote Procedure Call
RTT	Round Trip Time
SQL	Structured Query Language
TCP	Transmission Control Protocol
TPJ	Time per Job
UMTS	Universal Mobile Telecommunications System
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
W3C	World Wide Web Consortium
WHATWG	Web Hypertext Application Technology Working Group
WLAN	Wireless Local Area Network
WWW	World Wide Web
XHR	XMLHttpRequest

Acknowledgements

First of all, I would like to thank my advisor, Christoph Wieser, who has supported me throughout the process of writing to its completion with helpful advice and his know-how. He was enthusiastic from the very start of this thesis and contributed greatly to the development process. At the same time, he left a lot of freedom for own ideas and experiments. Beyond his expertise, his friendly and helpful manner made it a time that will stay in good memory.

I would also like to thank Prof. Dr. François Bry, whom I respect for his work and whose keen perception and constructive criticism led to relevant concepts for this thesis. His ideas and his interest motivated me, while his lecture about Web technologies inspired and enriched the entire work.

I want to thank all the open source developers, for putting their time and energy in the development of free programs. Their work made not only this thesis but also the whole progress in Web development possible. In this regard, I would also like to thank the researchers at the Teaching and Research Unit for Programming and Modeling Languages and everyone else involved in the ARTigo project for working on such an interesting and useful application.

Finally, I would like to thank all players for their participation in this experiment. I was very happy about the nice comments in the feedback and the challenges in snake- even though I was without a chance.

Philipp Langhans

Chapter 1

Introduction

The efficient collection and analysis of data is a relevant topic in many business processes. In addition, analyzing vast data sets is a key discipline for gaining new knowledge in a wide spectrum of scientific fields. The main challenges in both cases are frequently changing and fast growing data sets. Application fields that have to face these challenges are, for example, social networks, web server environments with large log files, sensor networks, military surveillance and Internet search engines.

To meet the increasing demands, companies in the respective areas spend billions[1] on improving and scaling their technology, especially their data centers. But even modestly successful startups have to deal with user bases comparable in population to nation-states [20]. The benefits of analyzing and understanding the collective intelligence in user bases or data sets are often more user friendly and improved products, resulting in more profit. For that reason, the importance of the capability to deal with large quantities of data can not be overestimated in the startup economy.

Market leaders like Google, Facebook, Amazon and Yahoo are able to fulfill the needs of millions of clients daily, due to their experience, specialized knowledge and forward-looking technology. Their approach to manage complex and growing problems in petabyte data size is a combination of customized file-systems, sophisticated network architectures, giant data centers and groundbreaking algorithms.

In contrast, companies and especially scientific research projects with limited financial means have serious problems to process simulations and calculations of high complexity on large data sets. Affected scientific fields deal for example with genomics, biological or meteorology data, medical records or history archives.

[1] In its earnings reports, Google reported \$1.9 billion in spending on data centers in 2006 and \$2.4 billion in 2007. [34]

Consequently, the need for scalable systems as well as accessible and inexpensive resources is very high.

This work aims to solve scalability problems in web environments by swapping computational tasks to connected web clients. The imbalance between service providers and consumers is thereby regulated and resources increase with every added consumer. The proposed approach could also be used, for example, to create a sort of overlay network on large networks like university or social networks in order to aggregate large clusters and to utilize the voluntary supplied resources to compute complex tasks.

The major contribution of this thesis is a flexible web framework that enables the computation of automatically parallelized MapReduce jobs in JavaScript with conventional web clients, that is, browsers. This work thereby addresses scalability problems and architectural challenges in web environments.

This thesis is structured as follows: Chapter 3 describes the current state of the art in web development or more specifically requirements for building a MapReduce framework, based on a given Internet infrastructure. The proposed approach uses a LAMP stack on the server, to provide low entry barriers for the deployment, testing and further development. Thus, chapter 3 focuses on the LAMP components, JavaScript and browsers representing the client side, but outlines principles of MapReduce as well. Chapter 4 describes the concrete framework implementation with its respective components for the distribution, management, calculation and aggregation of jobs. In this context, many factors and challenges are discussed that determine a successful deployment of such a system. Chapter 5 is focused on measurement and evaluation techniques. The framework prototype is live deployed and embedded in a testing system, in order to gather more information about the framework and to obtain reference values regarding stability, performance and scalability. In this tests, volunteers are invited to visit a prepared web page and play a browser game, while processing jobs in the background.
This thesis will demonstrate that the quality of connected clients is much more important than their quantity. For this reason, chapter 5 also describes characteristics and factors that determine the quality of an originally vague cluster of web clients. All results and findings of the preceding chapters are summarized in chapter 6 and compared with a conventional approach. In addition, the web framework is modified to work as 'hybrid' and to be able to process jobs without connected clients. Further suggestions for optimizations, such as an improved allocation strategy, combine and assess the main findings at the end of this thesis and provide an outlook on further developments and possible application fields.

Chapter 2

Related Work

Google's MapReduce programming model, first described by Dean and Ghemawat [4], was the initial basis for a vast number of related articles and implementations. MapReduce was designed to hide the complexity of parallelization from the original target related computation, while providing a simple and standardized scheme for large-scale computations. This work transfers core ideas and principles of MapReduce to a web architecture context and is influenced by the original framework and similar implementations. MapReduce is described in more detail in section 3.4. The lectures by Bry et al.[9] served as an introduction into the topic.

Further MapReduce based researches include possibilities of running several MapReduce jobs on a single machine. The Phoenix[32] project makes use of multi-core and multi-processor systems with global shared memory buffers, whereas the Mars[15] framework implements a system, optimized for Graphics Processing Unit (GPU) processing. Phoenix is inspired by todays trend to have multi-core computers, while Mars takes advantage of the fact that, due to GPGPU, performance benefits of GPUs becoming more productive. Both approaches lead to promising results, especially when combined, but suffer from being limited by the maximum hardware performance of a single machine.

In addition to these computation frameworks, several promising projects were triggered by the combination of MapReduce and databases. One project whose popularity is growing steadily is CouchDB. CouchDB is a document-oriented database written in Erlang that makes extensive use of JavaScript to access stored data [37]. MapReduce is particularly used to build views which are needed to create relationships and provide aggregation features since all data is stored without a schema. CouchDB already provides several predefined MapReduce functions for this purpose.

One very popular and sophisticated project that makes use of MapReduce is Hadoop[38]. It is very similar to the system described by Google and adapts not only the actual MapReduce model but with HDFS also GFS. Hadoop is written in Java and maintained by the Apache Software Foundation. Numerous extensions, frameworks and applications are based on Hadoop. It is widely used in practice, was awarded for its innovative contribution and largely motivated this project.

MapReduce frameworks have been developed in multiple programming languages but it seems to be a contradiction to turn a high speed, shared-nothing, enterprise solution into a centralized and supposed slow web application. For this reason, fairly little projects have tried to combine the advantages of both fields or to analyze their interaction.

The few considerable related works that involve similar aspects are from Grigorik and Ryza and Wall. Ilya Grigorik[14] outlined several factors and ideas for the development of a JavaScript MapReduce framework on his blog. While his approach seemed very worthwhile and gained notable attention, only few projects followed the idea to implementation. The MRJS framework described by Ryza and Wall[35] uses JavaScript for the computation of MapReduce jobs in the browser and is most similar to the proposed system. Ryza and Wall describe a completely centralized system running a Django job server on a virtual machine with Amazon EC2 instances simulating accessing clients. Their paper is focused on algorithmic evaluations and the discovery of suiting application fields for their system. Due to the use of Amazon EC2, performance aspects of vague clusters are, however, simplified.

Specifications, benchmarks and blogs of leading web developers played an important role to understand the limits and possibilities of modern browsers and web applications. Steve Souders' blog for example describes many current performance optimization strategies while John Resig's blog provides information on testing, benchmarking and improving JavaScript applications in general. Further blogs addressed specific implementation problems and are cited in the respective parts of this thesis.

Chapter 3

State of the Art

The Internet infrastructure plays a decisive role when planning to build applications upon it. Initially, the Web and its protocols were specified for text delivery and as a consequence were designed and developed for a different purpose in comparison to high speed enterprise networks. With its support for multimedia contents and a growing range of web applications the Internet changed its nature and evolved over time, but it still has lots of restrictions on building complex applications upon it.

The next sections will outline the main task-related factors, which are relevant for designing and building web applications like the proposed framework and will weigh the 'pros and cons'.

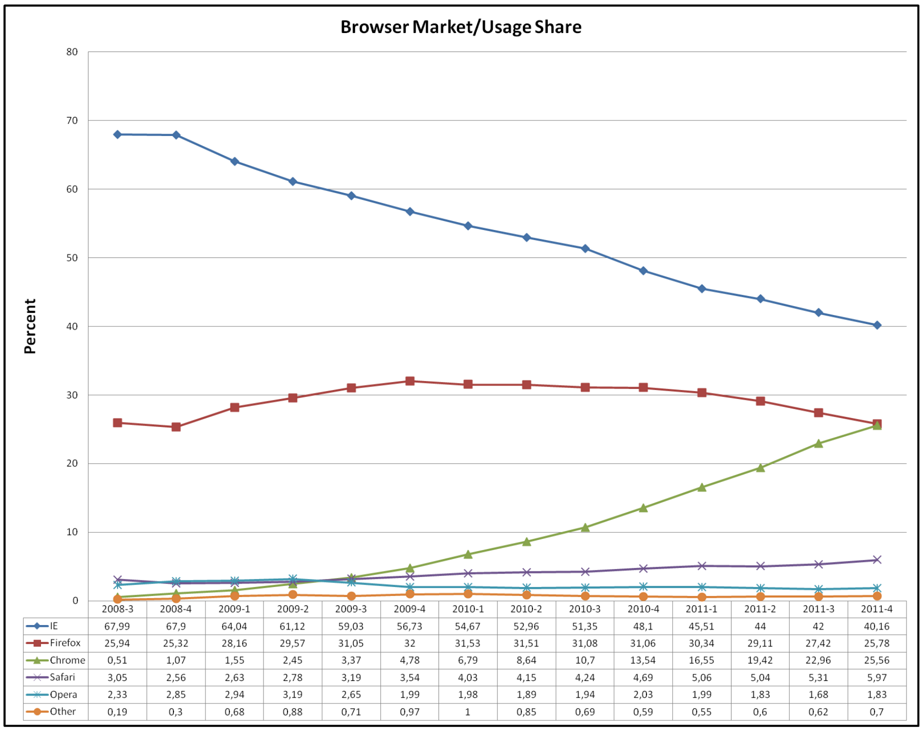
3.1 Web Clients

Generally stated, clients are defined as applications which access services provided by a server. In the WWW, web clients are users of resources, for example HTML files. Typically the transformation of an Uniform Resource Identifier (URI) into a rendered web page or the extraction of relevant data from a markup file is done by a web browser, but similar work can also be done by a crawler, bot or other kinds of specialized software. In the proposed approach browsers are no longer

Due to the fact that there is an ambivalence in the term 'client', the next sections will explain the interaction of the user, browser and JavaScript engine, acting together as a 'web client' in terms of client-server communication.

3.1.1 Web Browser

A web browser is a software application for retrieving, presenting and traversing information resources on the World Wide Web. This makes it one of the most-used programs on computers and its programming and establishment one of the most competitive markets in software development. On the one hand the so called 'browser wars' have led to a quick development which often means new features and higher performance but on the other have also led to incompatibilities depending on version number and producer.



	2008-3	2008-4	2009-1	2009-2	2009-3	2009-4	2010-1	2010-2	2010-3	2010-4	2011-1	2011-2	2011-3	2011-4
IE	67,99	67,9	64,04	61,12	59,03	56,73	54,67	52,96	51,35	48,1	45,51	44	42	40,16
Firefox	25,94	25,32	28,16	29,57	31,05	32	31,53	31,51	31,08	31,06	30,34	29,11	27,42	25,78
Chrome	0,51	1,07	1,55	2,45	3,37	4,78	6,79	8,64	10,7	13,54	16,55	19,42	22,96	25,56
Safari	3,05	2,56	2,63	2,78	3,19	3,54	4,03	4,15	4,24	4,69	5,06	5,04	5,31	5,97
Opera	2,33	2,85	2,94	3,19	2,65	1,99	1,98	1,89	1,94	2,03	1,99	1,83	1,68	1,83
Other	0,19	0,3	0,68	0,88	0,71	0,97	1	0,85	0,69	0,59	0,55	0,6	0,62	0,7

Figure 3.1: Browser Market Shares [36][1]

The figure 3.1 illustrates the market shares of the leading browsers. It should be noted that each of these browsers has up to 15 sub versions. Even if some vendors are forcing the user by time to a newer version, there are still significant differences in terms of functionality and performance among the earlier versions, which are actively used, and the latest stable versions or pre-builds. Moreover, the figure illustrates that new browsers penetrate the market, taking others shares or even displace them. This rapid change requires the developers to be highly flexible and durable to remain competitive.

[1] Values are estimated, based on [36] (see also [45] [11] [19])

When planning a client-side web application, the development scope is completely given by the browser. The reason for this is the browsers' responsibility for correct rendering i.e. content representation and successful script execution. For the latter purpose it provides the programmer with predefined interfaces (Application Programming Interface (API)s), enabling for example DOM manipulation or client-server communication over the XMLHttpRequest (XHR) object.

Incompatibilities depending on vendor specific API implementations are consequently the main problem in development and make it impossible to reach all Internet users with modern web applications. In addition many Internet users are using browsers that are outdated and simply do not meet the required functionality. This situation is intensified due to the fact that a browser change or update is not thought of as necessary by a significant number of web surfers. To sum it up there are users who simply do not update or change their browser resulting to them not being supported whereas other users update and alternate too frequently preventing an established standard. For this reason it can be a challenging and time consuming task to support as many browsers as possible.

While many people are using their browsers simply for the purpose of fetching websites, various new and innovative application concepts have arisen. This new generation of so called Rich Internet Application (RIA) is meant for a successive substitution of typical desktop software, and is therefore imitating the software's classic look-and-feel as well as its behavior. Furthermore the entire discussion about cloud computing indicates that the relevance of new software models and especially the role of web browsers as futuristic Operating System (OS)[12] or platform, is becoming of more importance.

It is safe to say that for many people a browser is no longer just a tool for querying pages. With the possibility to render complex graphics or play multimedia content, the browser performs important tasks in order to define an entirely new user behavior. By increasing their power, browsers encourage that the boundaries between desktop applications and web pages are blurred more and more and completely new application fields become possible.

Since all browsers speak (more or less) the same language, web applications are able to run platform-independent and linked (if not interconnected) on almost any number of computers worldwide. Even if some of the mentioned incompatibilities prevent users from access to these services and technologies, it is nevertheless necessary to write new and way leading applications. At the same time it can be an advantage that users do not make use of the potential of this new generation of browsers and thereby create huge potentials of unused thus usable power. These characteristics demonstrate that a browser is a perfect platform for new models and application fields. The possibilities are however further limited by the user and the JavaScript engines, as described in the following sections.

3.1.2 Internet User

The Internet is populated daily by approximately 2,095,000,000 people[25], which is about 30.2% of the global population. Thereby the average user spends 2 hours on active web browsing. Beside this active browsing, many users are connected to the Internet in a passive way e.g. for email synchronisation or short message transfer (push notifications).

Social media contributes a large share to this activity. For example, the average Facebook user spends about 7 hours a month[42] for connecting with people in his life. Although the statistic may be influenced by inactive and passive members, it is likely that the actual time frame achieved daily by active members is equal to that of one month.

The facts stated above are partially necessary for calculating the mass of potential clients and computational power. But more importantly they are crucial for the analysis of the human behavior, which determines the entire architecture and development process.

In particular must be pointed out that a browser is only a piece of software controlled by a human being. For that reason the browser, as a communication partner, is significantly influenced by the human behavior.

One example for this is that if the user jumps off a site the connection will be lost. Furthermore the user will prevent any actions if the performance or surf experience is limited in any way. The same applies to bandwidth limitations and costs. However, the most important aspect is data privacy and security since data is transferred to the client in plain text and program code is executed. All these factors must be analyzed in the approach as they massively influence considerations on failure tolerance, redundant data distribution, encryption and timeout duration.

3.1.3 JavaScript (Engines)

One core component of modern web browsers are the JavaScript engines. JavaScript is understood and executed by every modern browser that allows script execution and is used in 90.8% [47] of all (considerable) websites.[2]

JavaScript was designed in 1996 as interpreted language, therefore the engines were originally called interpreters. The performance of the interpreter has always been adapted to the computational tasks. These were composed primarily to support functions such as form validation or display alert and error popups.

[2]compared to 2005 50% [13] ; 2006 73% [26]

In the following decade the Internet grew massively and the demands on JavaScript became larger. With the introduction of Document Object Model (DOM) and the possibility of client-side page manipulation (DHTML) new application fields quickly became apparent. The new application fields, which were obtained for JavaScript, frequently entailed, that small scripts became large codes with thousands of lines. But for this type of use JavaScript was not ready yet. The results were slow or even crashing websites and the reputation of being slow as well as susceptible to lead to problems.

Additionally, JavaScript has offered and still offers some possibilities to abuse functionalities, such as forcing users to certain actions, keeping them on the website and exploiting security gaps in the browser. Due to the mentioned facts JavaScript is still blocked by many users[3] and was even considered to be useless and harmful.

Nevertheless, these outdated views disappear with the growing variety of application possibilities and usage. The real power behind JavaScript is well demonstrated and brought to the masses by modern frameworks like 'jQuery', 'Prototype', 'Dojo', 'Yahoo! User Interface' (YUI) or 'Google Web Toolkit' (GWT). Techniques like Asynchronous JavaScript and XML (AJAX) and Dynamic HTML (DHTML) thereby become easier to manage and are used in a large number of web projects.

Even if JavaScript runs through a continuous development process it still has some serious limitations.

The browser vendors respond to the problem of immature JavaScript interpreters and exploit them in their competition. There are dozens of benchmarks suites[4] testing the browser's performance in depth and acting as the driving force behind browser wars. This 'sideshow' is often called JavaScript engine race and leads to the development of high performance engines ensuring fast script execution, good benchmarks and as a result more users.

Figure 3.2 is an example for such a benchmark. In this case, the test was performed with and the Chrome browser with V8 engine did best. The V8 engine which was introduced in 2008 by Google, was the first engine with a just-in-time (JIT) JavaScript compiler. In order to keep pace with V8, Safari and Firefox followed with their Nitro and TraceMonkey engines. This marked the end of the interpreter era. Today, JavaScript is no longer interpreted but compiled into machine code resulting in code that is executed significantly faster.

At the same time, the usage of JavaScript is beyond client-side execution. Many projects and experiments like NodeJS, CouchDB or Rhino were initiated to build engines which can run JavaScript server-side or enable standalone applications.

[3]at the moment 2,124,689 noscript users: [22]

[4]for example Dromaeo, Kraken, Sunspider

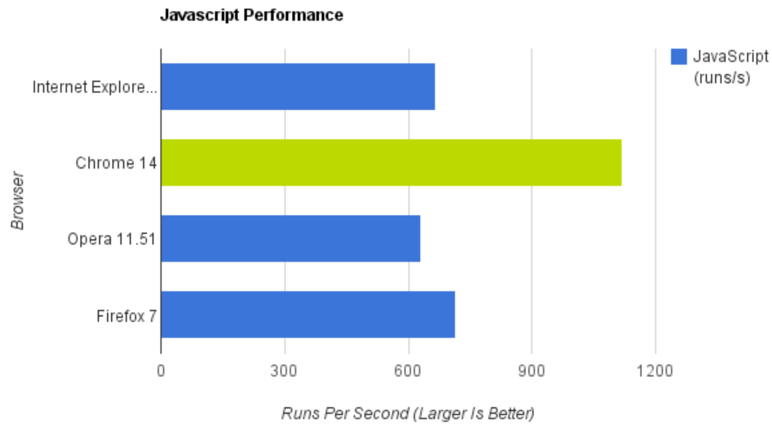


Figure 3.2: Dromaeo JavaScript Performance Benchmark [51]

But the greatest advantage of using JavaScript in browsers over most conventional languages and applications, is not least the absolute platform independence. JavaScript runs completely out of box and can be executed parallel to ones web page visits. The opportunity of having billions of potential clients without having to install a new software, a virtual machine, an additional plugin or anything else first while combined with sophisticated ways of distribution, is by far the leading argument to develop this framework in JavaScript and run it in the browser.

3.2 Web Server

Computers which store content and provide resources are called servers. Depending on the offered services, they can be further distinguished in mail servers, file servers, web servers and so on. That is why both the offered service as well as the physical machines are called servers.

Web servers and web clients communicate over HTTP. Servers, responding on HTTP requests are often referred to as HTTP servers. One major task of a server is to listen on open ports for incoming connection requests. If an HTTP request comes in at a server, it is interpreted and processed according to certain rules. Often, this processing is done by merging other resources like images, JavaScript files or CSS files into the same response. If the response is generated, it is called a dynamic page, else if only explicitly requested files are sent back it is called a static page.

For the dynamic response generation one particular system of corresponding components has established. The system is called LAMP stack and the name refers to the first letter of each used software component. These are Linux as Operating System, Apache as web server, MySQL as database management system (DBMS) and PHP as server-side scripting language. Major advantages of this bundle are that all components are freely available as open source software, widespread and thus continuously developed. LAMP made it possible for many web developers to create products and solutions that put some comparable enterprise solutions out of business, due to their performance, stability and price. The existing documentation and the beginner-friendliness, form an ideal base for prototyping while leaving the option to develop a scalable, consumer or even enterprise product.

In the following a brief description of the (L)AMP components and a couple of factors for a successful usage are given.

3.2.1 Apache HTTP Server

The Apache HTTP Server (Apache) is an open source web server software supported and maintained by the Apache Software Foundation. It is used to deliver resources which are requested over HTTP. With over 300 million different websites using it and an estimated market share of 65%[21], the Apache is the most popular HTTP server software in use.

The web server is responsible for processing client requests based on predefined criteria and sending the required data to the correct client. Mostly, this data consists of static HTML files but the Apache also allows the dynamic content generation with scripts.

The Apache web server pursues a modular concept, which makes it very flexible in usage.[40] Functions which are not part of the core must be extended with additional modules. Such modules can be integrated either dynamically or statically. Static modules are compiled directly into the server's core, resulting in a speed advantage over dynamic modules. Although every change effecting one of those modules requires a complete recompiling of the core. For dynamic modules it is enough to change the relevant module, resulting in less work for maintenance.

One example is the PHP module. In order to process embedded PHP source code, the required modules have just to be installed. Afterwards, the PHP files are handled correctly and processed by the engine. Another example of a module is mod_gzip. This module is build to compress data using gzip, reducing the size of the web pages and thus saving bandwidth.

In addition to these functional modules, there is another kind, called Multi-Processing Module (MPM). These provide adjustments to platform- and system-specific conditions. By the use of MPM various operating systems are supported more seamlessly and deeper settings depending on the type of website become available. The way in which client requests are treated is just one example. The MPM 'prefork' is activated by default on the used Ubuntu Linux, which means that depending on the used configuration the Apache forks a new process for every connected client. For the desired compromise between stability and performance, there are other modules such as the mpm_worker_module which implements a hybrid multi-threaded server. [41]

Beside those modules, the range of functions is strongly determined by directives, which are basically configuration parameters. The 'MaxClients' directive for example, sets the limit on the number of simultaneous requests that will be served. Any connection attempts over this limit will be queued, up to a number based on another directive.

The value which should be chosen for concurrent client connections is influenced by many factors. While some limitations may be bypassed by careful configuration, others are predetermined by the type of used software and hardware. The limitations may depend on type of request, type of content (dynamic or static) or used optimization techniques such as content caching.

If the MaxClients directive is set too small, the performance of the server is not used to capacity, and some queries are not answered. If in turn, the value is too big it may be that the generated child processes use the entire physical memory or crash the server directly. If the total RAM is used, the system starts to swap. To this situation, the documentation is cited as follows: 'A webserver should never ever have to swap, as swapping increases the latency of each request beyond a point that users consider 'fast enough'. This causes users to hit stop and reload, further increasing the load. You can, and should, control the MaxClients setting so that your server does not spawn so many children it starts swapping. This procedure for doing this is simple: determine the size of your average Apache process, by looking at your process list via a tool such as top, and divide this into your total available memory, leaving some room for other processes.' [39]

The Apache web server is not designed primarily for high speed performance but is nevertheless competitive against rivals and 'high speed server'. As mentioned before, one of the most important factors that determines the performance is however RAM since it will influence the number of possible child processes, i.e. simultaneous connections. Moreover, a fast CPU, a fast network card and a new version of the operating system are good to have for an efficient deployment.

The rest is controlled by directives and modules. These include in particular system-specific adjustments over MPM and and application specific settings like maximum execution time or max. clients.

The modular design of Apache is a major advantage. Nevertheless, an optimal individual solution requires more work. In addition, adjustments are largely dependent on empirical values in order to achieve the right compromise between stability and performance.

3.2.2 MySQL

MySQL is the most popular Open Source SQL database management system and is developed, distributed, and supported by Oracle Corporation.[31] It is used for enabling a quick and structured access to data. In addition, it allows adding, manipulating and processing various kinds of data.

MySQL is based on the relational database model. This means, different datasets are stored in tables or more precise in relations, which are mainly accessed through Structured Query Language (SQL). An important principle to avoid redundancy and anomalies is the normalization. As part of normalization, it is common to split logical records apart on many different relations. These different parts later have to be merged again using joins. The procedure is often criticized, since the so called segmentation is inefficient as well as obscure. In order to keep every record uniquely identifiable, additional artificial keys are often needed. Frequently, these do not bear to the specific application object and merely serve as a management information, but instead generate a lot of overhead. Again, this is another criticism of the relational model.

Even if there are some promising alternatives on the market with completely different models, MySQL still has a fixed place in the LAMP stack and can be installed seamlessly on most systems. For this reason it has great popularity and is offered and supported by many hosting providers. The documentation is very comprehensive and many forums exist answering the most upcoming questions.

One reason why MySQL has prevailed over the competition are probably the extensive PHP interfaces. The combination of Apache, MySQL and PHP is proven and constantly developed further. Even if MySQL has some disadvantages compared to other DBMSs, it is suitable for both small and very large projects with multiple distributed servers.

Depending on the field of application, the performance of MySQL with respective optimizations is regarded as very good.

3.2.3 PHP

PHP: Hypertext Preprocessor (PHP) is a general purpose scripting language[44] which is mainly used server-side for dynamic website generation. The comprised term 'Hypertext Preprocessor' indicates the use of PHP: First, the PHP code is embedded in a source document like a HTML file and interpreted on request. The block surrounded by the delimiters '<?php' and '?>' is then replaced by the computed result. Beside HTML, many other output formats are supported and possible.

This simpleness is one of the main advantages of PHP and leads to its good reputation. PHP's simple syntax and this minimal requirements enable agile development with multiple iterations. Another benefit of using PHP is the seamless integration of its modules into the Apache web server but it even can be deployed on most web servers.

Most of the server-side parts of the proposed framework are written in PHP. This makes the source code easy to read and to develop further but the main reason for using PHP is that it is the most used server-side scripting language with approximately 77% of all websites using it.[48] Due to this fact the entry barriers for testing the framework with new scenarios, environments and web projects are very low. Another benefit of this large user base is one of the biggest and most active development communities and therefore, many existing forums and documentations, as well as third-party libraries and solutions for common problems.

The downside of using PHP is originally akin to the one of JavaScript. Both languages are more or less designed to be interpreted and have to face resulting performance problems. In addition, the loose syntax can lead to bad programming habits and a bad code design. Since this problems are well known, some very sophisticated optimization strategies exist. Most of them deal with specialized compilers or accelerators.

The definition of a compiler is thereby broader than 'a program transforming source code into something executable'. In case of HipHop[5] the source code is transformed into C++ code and recompiled by e.g. g++ but such a compiler can likewise do some preprocessing and transform the source just into another representation in the same language.

Since PHP 4.0 the source code is however natively compiled on-the-fly due to performance optimization. The targeted output format (Zend opcode) is then processed by the Zend engine. To reduce overhead and avoid problems caused by repetitive on-the-fly compilation it is good practice to store the compiled codes in the server's shared memory. This acceleration technique is called opcode caching and is used in some very advanced solutions.

3.3 HTTP

The Hypertext Transfer Protocol is a networking protocol and is mainly used for transferring various data in the World Wide Web. HTTP is according to the ISO OSI model, a layer-7 protocol and relies on TCP/IP for web content delivery. Even if HTTP is mostly used to transfer web pages, it is not restricted to this use.

Communication between client and server over Hypertext Transfer Protocol (HTTP) is based on request-response message transfer. An HTTP request message is created, for example, by entering an Uniform Resource Locator (URL)

[5] PHP optimizer developed by Facebook

such as `http://www.example.de/page1.html` in the browser. Through this URL a web resource is referenced. In the example it is the file 'page1.html' on the server 'www.example.de'. In the next step the client establishes a connection to the specified server to request the file. The server is listening for such incoming requests on it's ports. Once a TCP connection is established the HTTP messages are exchanged.

HTTP has only two types of messages: request and response. Both have the same basic message structure consisting of 'start line' and 'header fields' and differ only in the additional 'body' field of the response which contains the data. In order to obtain a response, a request necessarily has to be preceded. The reason is that a system where a server is sending data to clients of its own accord, was considered to not be useful in the development of the early Web and its protocols.

Regardless of the two types of messages, HTTP has different transfer methods. These were intended to inform the server about suitable actions to be performed for the respective requests. The two most popular are POST and GET. GET is typically used for requesting a file from the server. POST transmits data to the server so that they can be processed further.

HTTP has some limitations that impede the development of complex applications. A major disadvantage was already mentioned which is the inability to send data to the client without a latter requesting of it.

Another disadvantage is the statelessness of HTTP. This means that sessions can only be realized through implementation in the application layer.

A fundamental problem of HTTP was and is the so-called 'slow-start' problem which is actually caused by the establishing of multiple TCP connections. The reason is that one HTML file can include several embedded external resources such as images or stylesheets. In HTTP 1.0 for each of these resources a separate TCP connections is established sequentially. The average top-100 site as of 2010 included around 40 of such additional requests per page.[24] With HTTP 1.1 it is possible to maintain a TCP connection open after successful HTTP message transfer which solves the problem. This is done via the 'kepp-alive' header field. Such persistent connections reduce latency by reducing the overhead of TCP. In addition, HTTP 1.1 uses pipelining which allows multiple requests to be sent without having to wait for the respective responses.

The above stated disadvantages are on reason why Google build its own transfer protocol named 'spdy'. Nevertheless, these disadvantages are of minor priority for the development of this framework, since the requests are of small number although they are of large data size. For this purpose HTTP should be well suited enough.

3.4 MapReduce

MapReduce is a programming model [...] for processing and generating large data sets.[4] It was introduced by Google in 2004 and since then adopted and implemented in many programming languages.

The name 'MapReduce' stands for both an approach to data parallelism and an implementation of a C++ library (developed at Google). The MapReduce framework was build to perform calculations on large amounts of data with many (inexpensive) commodity machines instead of a few high performance computers, in a reasonable amount of time.

The initiation of the MapReduce model by Google is motivated as follows. With an index size of common web search engines being around 20 billion pages (as of 2011)[23], the index computation is not feasible on a single super-computer. Since the Web grows significantly faster than the speed of (single core) CPUs increases, only the sequential reading of the Web would take months.[9]

"The purpose of MapReduce is to perform text operations and computations like PageRank in a massively parallel manner on very large computer clusters. To this aim, Google throws several millions of computers (with fairly standard hardware and software components) with a share-nothing architecture and connected via a high-speed network."[9]

The MapReduce model is in particular based on a master-worker architecture: The master does the preprocessing, data partitioning, data distribution, receiving of results, and the merging. The workers in turn process their assigned tasks independently. Workers can be master by themselves and have their own subworkers. The interconnected commodity machines are called nodes and grouped into large clusters.

The MapReduce development is triggered by the discovery, that related problems and computations often require the same algorithmic steps, and aims to specify and handle them. The approach thereby comprises two distinct steps, which are inspired by the *map*[6] and *reduce*[7] functions in functional programming.

The first step involves applying a map function to each key/value pair in the input in order to compute a set of intermediate key/value pairs:

```
void map(String key, String value):
  // key: document name
  // value: document contents
  for each word w in value:
```

[6]map is a higher-order function that applies a given function to each element of an array or a list

[7]reduce or fold is a higher-order function that recombines and modifies lists through a given combining operator. It can be best explained with a short example: applying reduce to an array like [3,2,1] with the + operator would result in 3+2+1=6

```
    EmitIntermediate(w, "1");
```

Listing 3.1: MapReduce Example: map[4]

The next step is to apply a reduce operation to the intermediate data to compute the final result:

```
void reduce(String key, Iterator values):
  // key: a word
  // values: a list of aggregated partial counts
  int sum = 0;
  for each v in values:
    sum += ParseInt(v);
  Emit(key, AsString(sum));
```

Listing 3.2: MapReduce Example: reduce[4]

The above listed example in pseudo code could be used to count the number of occurrences of each word in a large collection of documents. The user who specifies the job has to define the respective map and reduce functions which are called by the run-time system.

In order to illustrate this process in more detail, a concrete example is given below.

```
'How much wood would a woodchuck chuck
If a woodchuck could chuck wood?'
```

Listing 3.3: MapReduce Example II: Input

The map function in listing 3.1 emits for a given word the count of its occurrences (in itself), which is obviously always the number "1". Accordingly, the "document" in 3.3 would be processed to an intermediate structure such as listing 3.4.

```
'How':['1']
'much':['1']
'wood':['1','1']
'would':['1']
'a':['1','1']
'woodchuck':['1','1']
'chuck':['1','1']
'If':['1']
'could':['1']
```

Listing 3.4: MapReduce Example II: Intermediate Data

After applying the reduce function, which sums together all emitted counts the result looks similar to listing 3.5.

```
'How':['1']
'much':['1']
'wood':['2']
```

```
'would':['1']
'a':['2']
'woodchuck':['2']
'chuck':['2']
'If':['1']
'could':['1']
```

Listing 3.5: MapReduce Example II: Result

In the given example, one step is performed implicitly by the run-time system. The map function takes as input (key, value) pairs and generates a set of intermediate (key, value) pairs. Only if all values associated with the same key are grouped together in between, the map function results in sets of pairs of type (key, list of values).

After this grouping phase, the reduce function operates on these (key, list of values) pairs and merges the values to create a reduced representation, as in the example, a single aggregated value.

The example also illustrates that different parts of the input string can be processed independently without affecting the overall result. This behavior is typical for a data-parallel processing without side effects.

An important design aspect of MapReduce is to hide the complexities of parallelization and monitoring. This goal is achieved by building an additional and simpler abstraction level hiding details such as parallelization, fault tolerance, load balancing or localization. In order to specify a job for the MapReduce framework the user only has to define the map and reduce function and pass them with the data. Due to a swap in terminology the user-specified map function is just the argument for the higher order map function which is executed by the system and adapted from functional programming.

MapReduce is designed for the use in large clusters of commodity machines, so it is in principle a suitable solution for the use in a web architecture too. The model can be applied well to new application areas due to its customizable data granularity, its computation flexibility and the fact that the computation is done in several phases.
However, MapReduce is not suitable for all types of tasks. Classical database technology, for example, is more suitable when the data has a manageable size, and is rarely updated. However, if the indexing of the data is complex and a highly parallel computation could be faster, MapReduce is probably the better choice. A major advantage is that MapReduce jobs are often defined with a few lines of code, which allows a flexible testing of scenarios and calculations in order to determine characteristics of the cluster.
The data-parallel processing ensures independent partial results, making differences in speed or even processing failures easier to handle. Due to these characteristics, MapReduce scales very well and is well suited for the distributed web framework.

But there are also many differences in the network infrastructure and the availability of the clients. The undefined and inconsistent processing power and bandwidth, as well as hard or late detectable failures are expected to be the biggest problems in the development. The next chapter describes the used architecture and also differences in the implementation in more detail, while addressing such issues.

Chapter 4

Framework Architecture

Merriam Webster defines a framework as 'a basic conceptional structure (as of ideas)'. The next sections will describe the intuitive construction of a MapReduce framework for web clients built with widely used and known techniques and languages.

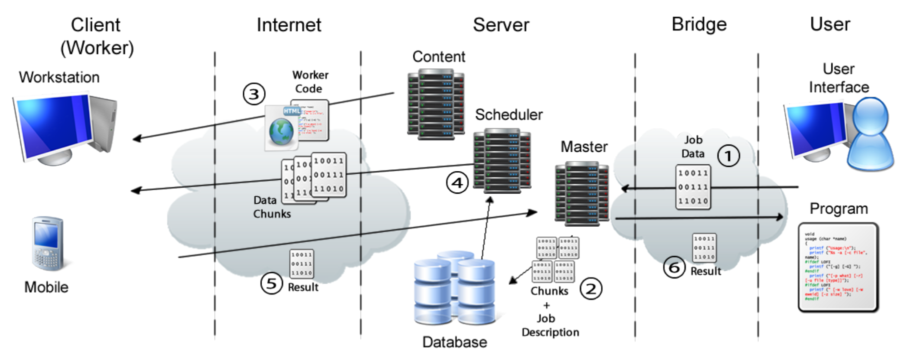


Figure 4.1: Framework Overview

The resulting web framework and its architecture is not equal to the framework described in 'simplified data processing on large clusters' by Dean and Ghemawat [4]. The fact of having a web architecture with different, predefined transport protocols, less performant workers, lower transmission rates and vague information about the cluster causes all instances to be defined basically different. The steps that are necessary for building a controlled master-worker network, are however related with the result that both frameworks share similarities but at the same time differ in large parts.

The main purpose for the development is to measure the web clients' performance in practice, provide an understandable and extendable system which can be run on various machines in todays Internet ecosystem, and to give a proof of concept. The following sections will describe each column of the architecture as it is illustrated with figure 4.1, from left to right, and therefore will start with the web workers (fig. 4.1 - Client) and related topics like inter process communication or MapReduce, followed by the scheduler as principal communication partner, the master as interconnected instance (both in fig. 4.1 - Server) and finally the bridge and user interface (fig. 4.1 - Bridge & User).

4.1 Worker

Workers are (web) clients or more generally stated, machines executing codes, which enable them to receive and process commands. In order to obtain data and perform the given tasks, they further have to be connected to the server over HTTP. The actual framework implementation is designed for using primarily web browsers as clients. A browser becomes a worker in this context by visiting a prepared web page, which contains the JavaScript worker-code. Simultaneously, the browser is (trivially) connected to the web server over HTTP.

Once the page content and the included scripts are loaded completely, the worker-code is executed. The worker-code consists of different parts that cover different design aspects and influence the worker's behavior. Some important aspects and techniques are outlined in the following subsections.

4.1.1 Concurrency

To be able to compute results in the background while surfing, JavaScript codes have to be executed parallel. Otherwise, if the computation would not be done parallel in the background but sequential, the user would have to wait until all jobs were done before he could interact with the Graphical User Interface (GUI) or webpage elements again and without troubles. The reason for this is that the JavaScript execution and the UI updates are both controlled by one single thread called UI thread. That means, as long as any JavaScript is running and therefore blocking UI updates the website is not responsive to user interactions. This scenario would be unacceptable under normal conditions and programming aspects, but has long been common practice. [28]

JavaScript is naturally single-threaded which means that there is only one script executed per window at one time. Since the calculations can be complicated and time consuming, this situation inevitably leads to one of the most fundamental problems in development. Even the creator of JavaScript, Brendan Eich, is quoted as saying '[JavaScript] that executes in whole seconds is probably doing something wrong'.[27] So, in order to be able to compute even time consuming tasks the execution needs a sound parallelization strategy.

AJAX JavaScript is single-threaded even if some techniques such as AJAX seem to be executed parallel. Actually AJAX is working with asynchronous events passed through an event queue. The basic idea of AJAX is to avoid waiting time between HTTP requests and responses[10] and thus be more user friendly. For this reason, most browser APIs allow the developer to send non-blocking requests to the server i.e. without active response waiting. The completion of a typical AJAX request is then indicated by the firing of an 'onreadystatechange' event by the requesting XHR Object[1]. The event handling is again done sequential even if the page load was done parallel to the actual script execution, which is no contradiction but the principal of asynchronous communication.

Splitting Depending on the browser, the mentioned event handling can be done differently. There may even exist asynchronous and synchronous events with different queues, but all browsers had in common that the only way to achieve a pseudo parallel execution was to break up a job in chunks and split their execution apart using timers[17] and events. This technique allows the browser short breaks in which it can process other incoming events or UI tasks and evades warnings like the so called runaway script dialog ('script execution is long-running').[29][16]

WebWorker API Since the technique of splitting is difficult or not feasible at all, there was a lot of effort to develop an alternative approach. The 'WorkerPool API' by the Google Gears Team was such an approach and also the basis for the 'web worker' recommendation. Due to internal disagreements, the recommendation was first made by the Web Hypertext Application Technology Working Group (WHATWG)[50] but is also published by the World Wide Web Consortium (W3C) as a part of the so called HTML5 specification[46].[49] With regard to the WHATWG specification and to avoid name conflicts the parallel executed scripts will be named in the following 'web workers' and are just a part of the worker role in the framework architecture.

The web worker API enables scripts running in the background independently of any user interaction with the interface. This allows the UI to be responsive and process events like clicks and typing on the keyboard without being interrupted and vice versa. The API is however not yet supported by all modern browsers as illustrated in figure 4.2. To avoid conflicts, the browser version should therefore be checked at the beginning of each site visit. Systems that are not supported then should display a short information instead of the UI.

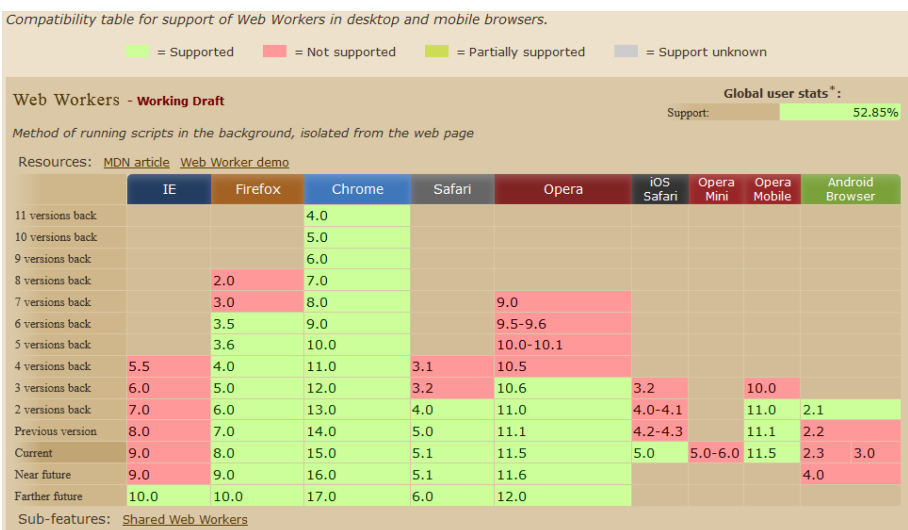
A web worker process is initiated with a new worker object instance as described in listing 4.1.

[1] The browser's XHR API can be used to transfer data from a running browser script to a server over HTTP

In line 2, a new web worker instance is created. For this purpose, the constructor is passed the path to a corresponding worker script, which will be executed parallel and independent. The following lines define event handlers that are necessary in order to communicate with the independent web worker. The communication process of the web worker is described in more detail in the next section.

```
1  function startThread(){
2    worker = new Worker("workerThread.js");
3
4    worker.onmessage = function(event) {
5      var statusCode = event.data.substring(0,3);
6      switch(statusCode){
7        case "100":workerStat['wnr'] = event.data.substring(4,event
             .data.length);break;//worker registered
8        case "200":workerStat['jobs']++;workerStat['size']=
             workerStat['jobs']*1607;break;//job complete
9        case "300":break;//Info
10       case "400":workerStat['error']++;break;//error has occurred
11     }
12   };
13
14   worker.onerror = function(error) {
15     workerStat['fatal']++;
16     throw error;
17   };
18 }
```

Listing 4.1: Web Worker Example (pseudo-code)


Compatibility table for support of Web Workers in desktop and mobile browsers.

= Supported = Not supported = Partially supported = Support unknown

Web Workers - **Working Draft**

Global user stats*: Support: 52.85%

Method of running scripts in the background, isolated from the web page

Resources: MDN article Web Worker demo

	IE	Firefox	Chrome	Safari	Opera	iOS Safari	Opera Mini	Opera Mobile	Android Browser
11 versions back			4.0						
10 versions back			5.0						
9 versions back			6.0						
8 versions back		2.0	7.0						
7 versions back		3.0	8.0		9.0				
6 versions back		3.5	9.0		9.5-9.6				
5 versions back		3.6	10.0		10.0-10.1				
4 versions back	5.5	4.0	11.0	3.1	10.5				
3 versions back	6.0	5.0	12.0	3.2	10.6	3.2		10.0	
2 versions back	7.0	6.0	13.0	4.0	11.0	4.0-4.1		11.0	2.1
Previous version	8.0	7.0	14.0	5.0	11.1	4.2-4.3		11.1	2.2
Current	9.0	8.0	15.0	5.1	11.5	5.0	5.0-6.0	11.5	2.3 3.0
Near future	9.0	9.0	16.0	5.1	11.6				4.0
Farther future	10.0	10.0	17.0	6.0	12.0				

Sub-features: Shared Web Workers

Figure 4.2: Web Worker Support[5]

4.1.2 Inter Process Communication

The worker communicates in three ways that differ fundamentally from each other, that is, the communication partner, the message flow and message type are different.

Worker - Web Worker: Event Driven The web worker API has predefined methods for communicating with a worker 'thread'. Using the postMessage() function, messages can be sent within the worker thread to its parent. Since the entire processing of jobs takes place exclusively in the web worker thread, the messaging is reduced to simple status updates without any data transfer. For the status updates, the framework defines four types of notifications. One message type is used for example to signal a completed job and uses the status code 200 followed by a brief human-readable message. The definition of these messages is listed in 4.1.

The main or UI thread registers for such status updates with an 'onmessage' callback (listing 4.1 line 4) and acts as an observer. Once the onmessage event is triggered, the bound callback is executed and the message type is checked against the given status code. In the next step the relevant functions are called, which handle , for example, the counting of completed jobs (lines 7-10). In addition to this regular communication, the API has some special events like the 'onerror' event in line 14, which is triggered by runtime errors in the worker thread.

To stop running workers, they can be 'killed' with the terminate() function, which aborts running operations immediately. Another, more gently, possibility to stop a running worker process is to self-define an exit routine. This function can be called, for example, if the user leaves the site or loses the game. For processing such UI thread triggered events, the worker needs to register an event handler as well. Once the web worker receives a message signaling it to terminate, it can teardown established connections such as job requests and close itself afterwards.

Worker - Scheduler: Server Push In order to get new jobs, the worker thread has to communicate with the scheduler (see figure 4.1 #4). For this purpose the worker uses parameterized HTTP requests. A worker does not have to know the concrete data that it is requesting in advance, which means that a standard request is sufficient at any time. Since HTTP is stateless and some conditions have to be checked server-side, the worker, however, should submit its registration number with every request.

The worker-scheduler communication seems to be apparently the simplest form of communication. Now however should be assumed that a worker is requesting new jobs while the system does not have pending jobs. As a consequence, the worker will have to wait for a certain time frame and afterwards repeat its request. In fact, this polling would work in some cases but would be critical for several thousand clients involved, sending requests multiple times per second. At the same time, the worker would be working to some extend and generating traffic without having to do so. Due to long waiting times, the situation would be intensified further and arriving jobs would not be processed promptly. The most efficient solution in this case would be if the scheduler informs the workers about new jobs itself - a concept also known as server push.

Since the HTTP protocol is request-response driven, an artificial back channel has to be created for such sort of communication. This channel is intended to enable the server to send push notifications like state or job information without preceding request[2]. Methods for achieving this behavior without polling the server frequently are encompassed under a programming model named 'Comet'.

The method used for the JSMapReduce framework is based on XMLHttpRequest long polling. In this context an HTTP request is not answered by the server until a desired event (i.e. for example the creation of a new job) arises. While receiving the response for one preceding request the client will immediately initiate the next connection causing some kind of dedicated line. The downside is that scripts like these have a very high running time which often requires a special configuration of the web server. The Apache for example must be configured in a way that allows many clients to be served simultaneously and scripts with a high 'maximum execution' and respective 'keep alive' time, which is not unproblematic at all.

```
<script src="js/comet.js" type="text/JavaScript"></script>
<script>
  var id = 1234;
  var mycmt;

  function getJob(){
    mycmt = new MyComet("scheduler.php?wnr="+id,doJob);
  }
</script>
```

Listing 4.2: Comet: Client-side (Worker)

```
<?php
while($i < $timeout){
  if(newJobs()){break;}
  usleep(1000000); //sleep for 1 second
  $i++;
}
?>
```

Listing 4.3: Comet: Server-side (Scheduler)

The listing 4.3 illustrates a minimum working example for the used server push technique. In this example the implemented Comet module does nothing more

[2] To be precise, even with the method used, a request has to be sent in advance

than submitting a GET request via XHR object to a given URL - in this case the URL of the scheduler script. The worker-number (wnr), sent for stated reasons, is the only argument and transferred directly via URL. [3]The second constructor argument (see line 7) is a pointer to the callback function, which gets executed for incoming responses. The received data is then passed to this callback function followed immediately by the initiation of the next (Comet) request.

As long as the worker's request is not answered a permanent or open 're-quest connection' is maintained to the scheduler. This not-answering in turn is achieved by the scheduler script by going into an infinite loop. The connection's duration can be controlled by a timeout variable (see line 2 in 4.3). If this timeout exceeds, the scheduler removes its open connections and answers the request with an error code for timeout. This situation is i.a. achieved in case that no new jobs are available.

The regular way for exiting the loop is however that jobs were found and are ready to be assigned (line 3). In this case the loop is left with a positive feedback signal and the data is transferred to the worker.

Worker - Master: AJAX Once the workers have finished their calculation, the results are sent back to the server (see figure 4.1 #5). In order to compress and standardize the transmission, the JSON format (JSON2 library, written by Douglas Crockford) is used as serialization and representation format.[3]

```
function sendJSONObject(data){
  if(!validate(data)){
    error++;
    return;
  }
  var dataString = JSON.stringify(data);
  myAjax.sendPostRequest("http://myserver.pms.ifi.lmu.de/master/
      master.php", {result: dataString}, function(res){});
}

var resultStruct ={
  wnr: wnr,
  profiling: JSON.stringify(functionCalls),
  result: JSON.stringify(result)
}

sendJSONObject(resultStruct); //transfer results as JSON
```

Listing 4.4: Result Transmission

Listing 4.4 illustrates how the results and additional profiling information are transmitted to the server by performing an AJAX post. The callback method is not used in this example but could be used to verify a successful transfer.

[3]Of course, the worker id can not be chosen arbitrarily (unlike the example), but is assigned within the registration process

4.1.3 MapReduce

The worker thread is responsible for the entire computation. In order to ensure a maximum of flexibility, all the required data as well as the methods for this computation have to be transferred to the worker. For this reason, an AJAX request is sent to the scheduler, whose task is to assemble the required data and transmit them afterwards. Once the data arrives in the client the doJob callback is executed (see listing 4.2).

```
1  function doJob(response){
2    postMessage("300 starting job");
3    //string,(functions) builder(JSON+'::::'+data])
4    data = builder(response);
5    //array INIT(string)
6    var chunk = INIT(data);
7    //array map(array MAP(array))
8    var il = chunk.map(MAP);
9    //array GROUP(array)
10   il = GROUP(il);
11   //array map(array GROUP(array))
12   var result = il.map(REDUCE);
13   var resultStruct ={
14     wnr: wnr,
15     profiling: JSON.stringify(functionCalls),
16     result: JSON.stringify(result)
17   }
18   sendJSONObject(resultStruct);
19   jobcounter++;
20   //predefined job limit per worker
21   if(jobcounter>=1000){
22     mycmt.teardown();
23   }
24   postMessage("200 job complete");
25 }
```

Listing 4.5: Job Processing with MapReduce

In order to relieve the server, the input for the doJob method is a single semi structured string containing all the relevant data as well as the functions that are needed for processing. In order to process the data, the builder function, first, has to separate the functional area from the data area using a stop sequence (see line 4).[4] In the next step it is checked whether the 4 main functions are already defined, or whether they must be extracted from the functional part of the JSON formatted response. The 4 main functions for the MapReduce computation are INIT, MAP, GROUP, and REDUCE which are capitalized for a better distinguishability.

First, the init function gets the data part of the response as input. Most of the preparatory work dealt with computations on texts. For such tasks the init step can, for example, consist of splitting a large text on every newline sequence and

[4]For reasons of clarity, the respective function signatures are commented in the listing

to push those separated parts into an array afterwards. Nonetheless, whatever the init method does with the data the result should finally be an array.[5]

After a suitable structure has been created the map function can be applied to it. The higher order map function is called on an array and gets a function as argument. The passed function is called in turn for each element with the respective element becoming the argument. After the initiation phase the data should be present as array again. A simple example for the map procedure is shown in listing 4.6.

```
var sq = [1,4,9,16];
var rt = sq.map(Math.sqrt);
//rt is now: [1,2,3,4];
```

Listing 4.6: Map Example: Square Root

As mentioned before, the map function is a higher order function. The map of the MapReduce paradigm is the input function for the functional map and has to be written for each job by the framework user. In the upper listing this function is named 'MAP' and is, in the beginning, embedded in the response data.[6] After the map step the resulting array stores the intermediate data.

In many cases, the intermediate data has to be processed further before the reduce function can be applied to it. In this implicit performed step, the framework groups together all entries of the intermediate data associated with same keys. As can be seen in listing 4.5 the intermediate data is not sent to the server in the meantime but kept in the client. As a consequence the two phases of map and reduce are not clearly separated- all steps are performed in a single run.

After all values have been grouped by same keys, the reduce function gets called analogously to the map by passing it to the functional map. Only if all steps have been completed, the results can be serialized and finally sent to the server.

4.2 Scheduler

The scheduler checks periodically whether new jobs exist that are to be processed. If this condition is fulfilled the scheduler fetches all the required data and sends them to the clients.

4.2.1 Data Structure

As soon as new jobs enter the system, the master writes the corresponding entries in the database. These records contain all required information about

[5] or at least an object

[6] The difference between both maps is also described in section 3.4

each job and enable the scheduler to detect and handle new jobs in several interdependent steps.

For each incoming job request, the scheduler, first, checks the database entries for jobs that have not been processed yet. The corresponding table is called 'jobs' and is structured as follows:

jobID	cstatus	mapfunction	reducefunction
1	0	map(...)	reduce(...)

Table 4.1: Jobs Table

The 'cstatus' attributes in table 4.1 describe whether a job has been processed completely. As described in section 4.1.3, each job entry must further contain a corresponding map and reduce function. They are both just stored once for a job together with the job-id and the completion status. The raw data on the contrary is stored in a separate table and is only retrieved when a corresponding uncompleted job can be found. Priorities are not supported yet by the system. The scheduling strategy is First-come, first-served (FCFS) which means the particular job is executed that first entered the system.

The mere structure of the raw data is stored in the 'schedule table'. This table is used to associate each job with several smaller units of data - in the following referred to as chunks. The individual jobs can be associated with chunks through the 'jobID' field.

jobID	sID	astatus	cstatus	start	end
1	0	1320486173.221.27	1	0	9999
1	1	1320486173.221.27	1	10000	19999
1	2	1320486173.221.27	1	20000	29999
1	3	1320486173.221.27	1	30000	39999
1	4	1320486173.221.27	1	40000	49999
1	5	1320486173.221.27	1	50000	59999
1	6	0	0	60000	69999

Table 4.2: Schedule Table

In the preceding tests, the input was mainly made of simple texts. For this reason, the boundaries(start,end) are interpreted as line indices in text files. The boundaries, however, could be used to describe offsets of other file types as well.

Beside those partition boundaries, each chunk, again, has its own 'assigned'- and 'completion'-status. The assigned status informs if the data is already assigned to a worker(in progress) and if so to which one. The completion flag in turn is used for checking whether chunks are processed completely. The 'sID' is used to uniquely address each chunk in the context of one job.

If the scheduler detects a record that is neither assigned nor completed yet, the assigned field is set to the worker's id and the related data is retrieved from a table containing the raw data, as defined by the boundaries:

chunkID	line
1	xx.xxx.16.55 - [Thu Sep...2011] - "GET /seite1.php" - "Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/535.1 ... Safari/535.1"
2	xx.xxx.35.78 - [Wed Dec...2011] - "GET /snakeServer.php?request=highscore HTTP/1.1" - "Mozilla/4.0 ... Opera 9.24"
3	xx.xxx.55.37 - [Thu Jun...2011] - "GET /seite6.php" - "Mozilla/5.0 ... Gecko/20100101 Firefox/6.0"
4	xx.xxx.34.8 - [Mon Sep 12 13:03:55 CEST 2011] - "GET /seite1.php" - "Apache/2.2.14 (Ubuntu) ..."

Table 4.3: Chunks Table

The 'chunks table' stores the raw data. In the presented excerpt one record is used for each line of a web server log file.

Having the tables as described above, the scheduler would first select the job with jobID 1, since this one is not completely processed yet. In the next step, the partition (schedule) table would be searched for the corresponding entries. In this example, the partition with sID 6 would fit the requirements as it is neither assigned nor completed. The respective chunks are described by the start-end interval, so that in the last step all lines with chunkID 60.000-69.9999 can be sent to the client along with the map and reduce function.

Besides the mere distribution of jobs, the scheduler may also performs other functions. Authentication and load balancing are likely examples for such tasks and are described in the following subsections.

4.2.2 Authentication

The authentication ensures that only those workers receive jobs that are actually allowed to. Thus, the system is robust against errors and manipulation attempts. To authenticate workers, all registered workers are managed in a separate table. In the implemented system, only workers with a valid id are allowed to process jobs and receive information.

Moreover, heuristics which check cookies, IP addresses, user agents, OS and screen resolution are able to reliable determine workers independently of their purport worker ID. It should be noted that users must be informed in this case about the type of information storage and the appropriate privacy policy, since massive storage of personal information is forbidden in some jurisdictions. For this reason the implemented framework does only support the storage of a worker number without additional information.

Due to authentication, the system can be improved with little effort and especially data integrity is strongly enhanced.

4.2.3 Load Balancing

In order to relieve workers, the system needs a sophisticated load balancing strategy. Since worker scripts can be executed on almost any device running a modern browser, there is a wide diversification of supported devices and thus potential workers with individual needs and performances. Due to the fact that there is no clear restriction to devices, the framework has to decide on its own which ones are supported or refused. On the one hand, this situation leads to the advantage that a vast number of workers can be easily aggregated to large clusters but on the other hand leads to the disadvantage that the system would have to deal with a wide performance and bandwidth spectrum.

Moreover, it must be taken into account that modern smartphones certainly will be able to perform calculations but therefore need individually sized data chunks. Reasons for this is are e.g. the built-in hardware and network connectivity. Even if many smartphones use local WLAN access points whenever possible, they are only partially efficient in mobile usage due to UMTS and GSM (EDGE). Even more critical is the fact that many mobile phones have limited data plans. If a certain data limit exceeds, this may causes high costs for the user. In order to minimize this risk for the user and to guarantee a certain performance quality most mobile devices should be considered as incompatible to the system.

Another difficult device category are sub-notebooks such as tablet PCs and netbooks. Although these are more powerful, they are akin to smartphones in many ways, in particular caused by their mobile usage. Accordingly, a successful load balancing depends on the ability to classify and assign different devices to a suiting category that consequently receives individual data chunks, adapted to the hardware specifications.

To sum it up, a crucial problem in load balancing is characterized by the vagueness of potential workers. At the latest, when it comes to data partitioning, the respective chunk sizes should be definite. Are they too small, the risk of generating too much overhead increases. Are they however too big, potential workers will be excluded or even more critically they show unpredictable effects or crash due to excessive requirements.

For the stated reasons, it is necessary to determine reasonable data sizes for different types of devices based on empirical results and statistics.

One possibility for enabling different chunk sizes without risking a stable system is to use varying boundaries while keeping the raw data partitioned in high granularity. Done so, the worst case scenario is that the time, won by supporting as many clients as possible, is not worth much and due to complicated partitioning algorithms the preprocessing of each chunk lasts too long. This procedure, however, would at least allow to test a pseudo-adaptive solution for the load balancing.

The current partitioning algorithm creates a database entry for each line of the input text file, making it theoretically possible to vary the chunk sizes during execution. Whether this procedure is sufficiently efficient will be discussed in later optimization considerations.

4.2.4 Worker Failures

Delays and errors caused by worker failures are one of the the main problems when it comes to the development of up-scaling and distributed systems like this. The importance of such problems is pointed out in the paper 'MapReduce: Simplified Data Processing on Large Clusters' by Jeffrey Dean and Sanjay Ghemawat. Due to the use of a web architecture, the described framework is, however, confronted with this problem at an entirely different level. Only to make statements about error types and failure rates, extensive analysis and tests are needed in advance.

Compared to web architecture dependent reasons, hardware failures can be mostly neglected. Since the runtime of the worker scripts is coupled to the browser's JavaScript engine, the mere closing of a browser tab or window can immediately terminate an established connection along with the worker thread. Even more troubling is the fact that a worker can be terminated at any state by leaving or simply refreshing the corresponding website.

One approach for detecting and handling these failures, which has a low complexity but leads to good results, is simplified as follows:

1. For each received result check the corresponding chunk for preceding chunks that are assigned but not completed yet
2. Set their assigned status to false
3. Distribute the latest jobs directly redundant, since they have no relevant successors

This approach is very efficient if the overall worker quality is high and balanced, since in most cases the system must evaluate only one additional field. Further, should all chunks be of same size. The redundant distribution of jobs, especially to very fast workers, ensures an even and controlled processing. Jobs that are computed slowly or not at all, can be quickly detected. They are either distributed redundant or completed within a short delay.

A more precise yet complicated approach would be to monitor timeouts or distribute random jobs redundant from the beginning.

Nevertheless, most suited approaches have in common that they heavily rely on re-execution to deal with failing workers.

4.3 Master

The master is the coordinating instance between worker, scheduler, and user. For this reason, the tasks of the master are mostly management oriented such as job initiation and ongoing control.

The main task of the master is to receive jobs from the user and to perform a preprocessing. One substep in this preprocessing is, for example, the validation of data which includes checking for completeness and accepted format. The master, moreover, verifies whether the user is authorized to commission a job. Following these formal steps, the partitioning and meta data creation is done according to predefined specifications. Consequently, the master creates all database entries for the scheduler. While the scheduler only forwards data after a simple scheme, the master should ensure that this data is complete and consistent.

At the same time, the master manages the registration of new workers and administration of corresponding tables. In order to monitor and control other processes, the master communicates with them and defines interfaces to the system. The following parts will describe the relations between the master and those processes.

Worker - Master To get job order, the worker, first, has to register to the master. For this purpose, it sends an HTTP request asking for an ID or more precise a worker number(wnr). If the request for registration gets accepted, a record is created, containing all information regarding the workers' session. By evaluating the related entries for a certain wnr, one can determine how many jobs a worker has already processed. The wnr can also be blocked, which means that no more chunks will be allocated to the associated machine.

After a worker has finished its calculations it connects to the master again, to send results. If necessary, the worker is then removed from the system. The results are stored afterwards with their respective chunks.

Master - Scheduler The communication between master and scheduler is unidirectional and indirect achieved by using database entries. The master prepares all relevant database records while doing the preprocessing whereas the scheduler accesses them read-only. If the master wants one particular worker to not process further chunks, it simply changes the respective record in the worker table and the scheduler will block this worker from further allocation.

Master - User The communication between user and system is the most extensive and is described in more detail in the following subsection.

4.3.1 Bridge and Interface

Bridge and Interface are the access point for users and programs in order to initiate new jobs and monitor them.

Bridge In order to receive new jobs the frameworks commands a bridge-server written in PHP. The configuration of the bridge server is done in a config file. The server can be bound to a port and permanently listens for upcoming requests. The main loop of the server is described below in listing 4.7.

```
while($i<$timeout){
  // read client input
  $input = socket_read($spawn, 1024) or die("Could not read input\n");
  //get command part
  $command = trim(strtok($input, " ,|\n\r"));
  //get argument part
  $argument = trim(substr($input,strlen($command),strlen($input)-strlen($command)));

  if($command != ""){
    //notify user
    $bridge->respond("processing: " .$command);
    //look for RPC
    $fun = processCommand($command);
    if($fun!=null){
      //execute
      $fun($argument);
    }else{
      $bridge->respond("unknown command in: ".$input);
    }
  }else{
    $bridge->respond("no command was found in: ".$input);
  }
  $i++;
}
```

Listing 4.7: Bridge Server

All in all, the bridge server is kept fairly simple. Basically, it receives command strings from the user or from a program and executes the respective Remote Procedure Call (RPC). For this purpose, the entire input, first, has to be read. Afterwards, the command and argument part have to be separated from each other. Between these steps, the bridge's respond method is used to providing feedback over the running task to the initiator.

Server and protocol are independent. The mapping of commands and functions is realized through the processCommand method.

RPC Protocol The commands sent to the bridge are similar to shell commands. The string 'help -f optional', for example, would be decomposed by the bridge into a command- ('help') and arguments-part ('-f optional'). The processCommand method is used to check whether a command, such as 'help', is valid. Every command is defined and managed in an associative array, consisting of command strings and function pointers (command-actions). If a command string can be matched, the corresponding function pointer is returned. The associative array, together with the defined command-actions, and methods to resolve strings into function calls define the protocol.

Once a function pointer is returned, it is either called or, if null is returned, an error message is forwarded to the user (see lines 14-19 in listing 4.7. The individual command-action is then passed the argument string. Every command-action in turn has to parse all arguments from the string similar to the getopt method in C. Following, the respective operations are performed according to the set options. Such operations can be, for example, the creation of a new job or a data structure, the filling of an existent data structure from a file or the distribution of jobs or parts of data structures to connected clients. With each executed function, the user is given a specific feedback about running or pending operations and their status.

Interface In order to simplify the transfer of such commands and to encourage the implementation of hooks in Java, a bridge client was build up in Java. In addition, an interface together with a view was written, enabling the comfortable sending of predefined or individual commands. The GUI itself is divided into two views - the terminal and the monitor. The terminal is used to write and send commands and to analyze, filter and output response messages. The monitor is used for visualization of jobs and their progress.

Figure 4.3 shows the terminal view. The terminal, first, has to be filled with all information, required to connect to the bride server (ip, port). By clicking the connect button, the bridge establishes a connection to the bridge server and forwards the response to the terminal. The first response should contain a list of all available commands (see 4.3).

Once, a connection is established, the user can send commands to the bridge server. In the example the user sent the command 'status' with the arguments '-j 1 -t 90'. This particular command is used to request a status report for the job with id 1 for the next 90 seconds. The console is then put into a passive mode and only parses every response message as long as data is sent to the interface. Simultaneously to this passive mode, the monitor is started.

The monitor panel is shown in figure 4.4. It is used to visualize the progress of running jobs. The total size of a job is indicated by the gray area with each gray-filled bullet representing one chunk. Each connected client that is processing a job is displayed blue. All processed chunks are filled green and all faulty or canceled chunks are indicated with red. Additional information about which worker is processing which chunk and the worker's specification are possible but were initially neglected due to their influence on the runtime.

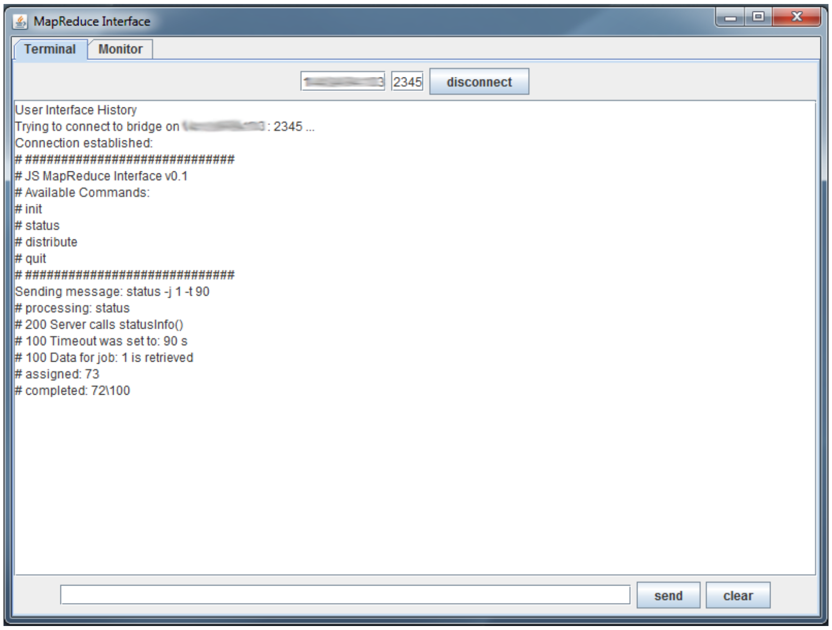


Figure 4.3: Interface GUI: Terminal

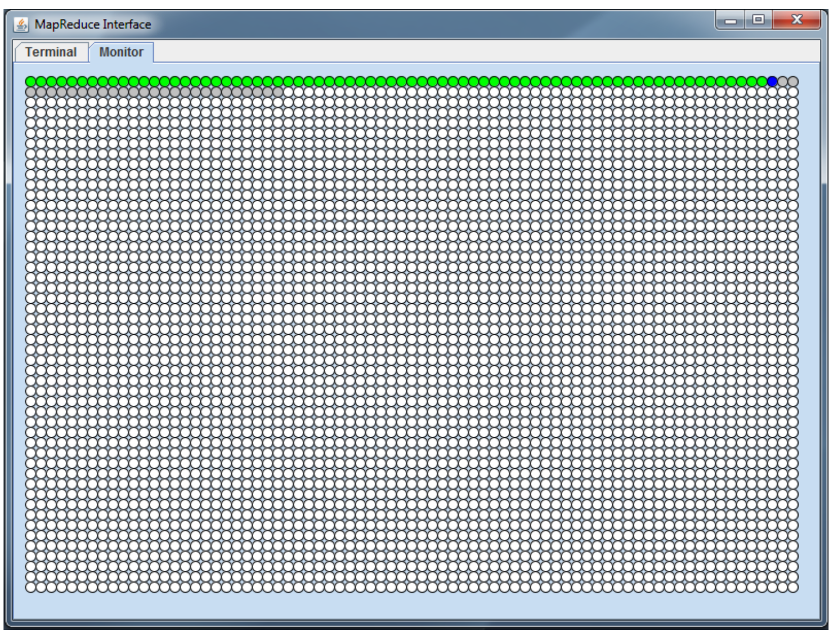


Figure 4.4: Interface GUI: Monitor

Chapter 5

Analysis, Profiling and Evaluation

One major task in the development of the JSMapReduce framework is the evaluation and performance measuring. Since many parts are written intuitively and only for testing purposes, they have to be examined thoroughly before further development iterations can be started. Without such measurements it would be uncertain whether and how much computing power can be gained effectively and whether the effort is worthwhile. Moreover, are these tests and evaluations needed, for proving the applicability of the framework and its core architecture.

At several points of this thesis was noted that a successful implementation of the respective modules depends heavily on experience and empirical data. One reason for this is the user which plays an important role and whose behavior affects design aspects of core parts of the implemented system. For this reason, the tests are designed such that a comprehensive picture of both, user and system emerges.

The first step of the evaluation is an analysis of the used components, the expected runtimes, and the detection of bottle-necks. In the following, a test is performed under actual conditions, that is in live-mode. The final step is to compare those results with the expected values. Additionally to the mere performance measurement the user's behavior and opinion is evaluated as well.

In order to draw comparisons, the test is not only performed with the framework but also a conventional approach.

The chosen example task, is to analyze a large web server log file. More precisely, it's about finding out the number of requests made for each month.

Step	JSMapReduce	Java
Data generation	-	-
Write data	- Read data from file Partitioning Write to DB	- (Write to file)
Read data	Read from DB Response generation HTTP transfer Receiving data on client side	Read from File
Processing	Data structuring Map Group (Intermediate data transfer...) Reduce HTTP transfer	Processing
Write results	Read from DB Aggregate result Write to file (Transmission to User)	Write to file

Table 5.1: Comparison I: Needed Computational Steps

5.1 Estimation

5.1.1 Analytical Considerations

The chapter 'State of the Art' (see page 5) has already pointed out many basic problems of the used web components and techniques. Now, these are considered in detail for the created system and its deployment. An appropriate approach to identify difficulties in this context is to analyze the life cycle of a suiting test case.

Table 5.1 illustrates the main, most time-consuming steps in chronological order, for achieving an equal computational goal with two different approaches. At the same time is demonstrated that the effort, needed for parallelization, is many times over a simple, single threaded computation. Additionally must be noted, that the performance of the connected clients is vague and the used web-techniques are in most cases less performant. It is expected, however, that a constellation exists at which the generated overhead pays off and the parallelization allows a faster computation. In order to clarify the most relevant factors, affecting the runtime, the next part will outline and estimate the needed effort for each step.

The following analysis is, nonetheless, only a vague prediction, which is aimed to compare the expected execution times for both programs run on the same machine and with respect to an increasing data size. The main goal is to make assumptions about the runtime behavior of some interesting parts and to extrapolate the results afterwards. If later tests can show that the the estimated values are sufficiently accurate, further predictions can be made on this basis aimed to find and optimize relevant and complex parts in arbitrary computations.

Reading and Writing Data The initial writing or data generation process can be ignored, since data should be already available in text form. This condition is the same for both approaches and fulfilled, because the Apache web server, for example, writes its data per default into a log file over a long period of time.

The text format should be sufficient to work with for a Java program, whereas the JSMapReduce framework requires the data in a more controllable and flexible data structure. For this reason a relational database like the introduced MySQL is more suitable. To write the data into the database, further operations are needed. First, the data must be read from the file, in order to enable the master script to iterate over all entries, partition them and write them into the database.

The duration of the read and write operations is determined considerably by the type of storage medium. In general, disk accesses are among the slowest operations due to the slow physical components in hard drives. The following read and write values, obtained with hdparm, describe the hard disk performance of the used virtual machine:
Writing: 85.6 MB/s
Reading: 216 MB/s
Without going too much into detail on rotation speed and head movements, the values are linearly extrapolated up to 1 gigabyte as follows.
Writing: 1024 [MB]/ 85.6 [MB/s] = **11.962 s**
Reading: 1024 [MB]/ 216 [MB/s] = **4.740 s**

Partitioning and Metadata The used partitioning algorithm reads the text file sequentially and writes each line directly into the database. This kind of preprocessing is quite simple and allows, due to the high granularity, more flexibility but bloats the payload with additional metadata which in turn influences the time to write the data.

Each line of the test log files has an average size of 161 byte.
Thus, a 1 GB big log file has about 6,669,204 lines.
With 4 bytes for each row-id an additional overhead of 25 megabytes is generated, only for identification purposes.(sizes according to [30])
With a chunk size of 10,000 lines per chunk and additional 256 bytes per chunk an extra 0.163 megabytes overhead is generated for the chunk management.

In addition to the mere data from the log file, approximately 25.5 megabytes of metadata is created which causes an additional write delay of 0.3 seconds. Since this data size is much less than a thousandth of the overall size, the generated overhead can be **ignored**.

Metadata is essential for the functioning of the system and simultaneously ensures a high flexibility. Even if the extra iteration for partitioning and the fact of writing the data twice is expensive from a performance perspective, it may be worthwhile in the long term or with increasing processing complexity.

Data Transmission Probably the greatest amount of time is spent during the transmission of data between server and client. The transmission is done via HTTP respectively TCP/IP. In this process, the data covers long distances with various delays. Sorts of delays are, for example, the processing delay which occurs in all transit systems, the queuing delay which varies with the load of the network, the transmission delay which depends on the size of the message, and the propagation delay which is physically dependent on the conductor material. All delays are encompassed by the terms network delay or latency.

The network latency can be expressed by the round-trip time (RTT), which is the length of time it takes for a packet to be sent to the destination host and the acknowledgment of that packet to be received. Latency, thus, depends on a combination of various factors such as spatial distance from client to server, the number of transit systems the message passes, the number and characteristics of links, the respective delays plus delays caused, for example, by network management duties. In contrast, does the RTT not include the processing time of the destination host.

A second, important factor for the individual perceived network speed is 'bandwidth' or more precise throughput. Both terms are often used equally, referring to bit-rate measures, representing the average rate of data delivery over a physical or logical channel per time. One reason for this terminology can be found in the Shannon-Hartley theorem which defines a proportional relation between a maximum data rate and the bandwidth in hertz of a communication channel in the presence of noise[52].

Thus, latency sets a lower boundary for a data packet to be transported from A to B, whereas throughput regulates the amount of data that can be sent within this time frame.

In general, a Internet Service Provider (ISP) assigns and regulates the bandwidth rate that is available to its consumers. Thereby, the achievable transmission rate of a connection is directly regulated by its costs. At the same time does the provision of a certain bandwidth not guarantee that a specific speed is actual achieved[1].

[1] This is caused by the fact that bandwidth and speed are not the same

With so-called 'bandwidth tests', one can measure the own data rate[2]. Some providers of such tests share statistics for comparison purposes too, if often with vague information. This information, is used in the following, to approximate the average network 'bandwidth'.

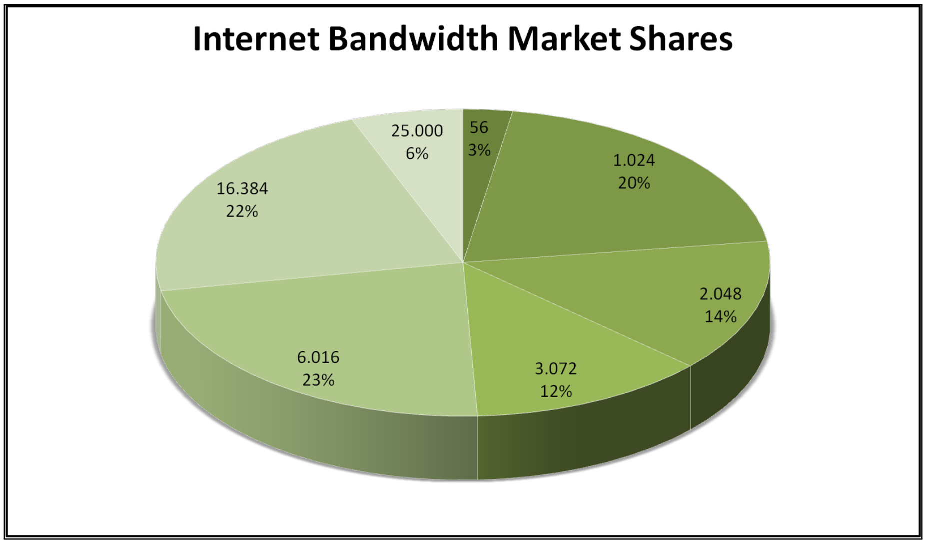


Figure 5.1: Bandwidth Market Shares [2]

The pie chart 5.1 illustrates the individual market shares of different data rates (bandwidth) according to the 'COMPUTER BILD' DSL speedtest'. All values are given in kbit/s and do only describe a maximum throughput. According to the OECD, the average maximum data rate in Germany is 17 Mbit/s while nearly 45 percent have Internet access with 2 Mbit/s to 10 Mbit/s in dowstream.[1] The 'DSLWEB' speedtest comes to similar results as the 'COMPUTER BILD' speedtest with the information of an overall measurement count of 174,135 in the last 6 months and average values for the actual achieved throughput per bandwidth rate(see 93).[8]

Based on the given values the average, maximum down-rate is assumed to be about 7,500 kbit/s ($\approx$ 937.5 kB/s) which would mean that the transfer of one gigabyte would require at least and without additional delays an estimated time of **1118.48 s** (18.64 min). Under same conditions the transfer of one chunk with 10,000 * 161 bytes = 1576.96 kB would take about **1.682 seconds**.

The statistic is related to the German broadband network which encompasses approximately 26.9 million fixed-line connections[6]. In addition to these fixed-line connections, many mobile devices such as smartphones have wireless Internet access but should not be considered further yet (see page 31).

[2] In fact, they often measure the 'goodput'

The statistics demonstrate the framework's strong dependence on exceptionally fast broadband connections. With a single worker that is gradually supplied with new jobs only the data transmission would take more than 18 minutes despite the fact that this procedure does not even takes place in a regular, single threaded computation. The data transfer is by a factor of roughly 230 greater compared with the reading from hard drive.

In addition to the input data, the generated intermediate must taken into account as well. The intermediate data could be sent back to the server, to be stored and scheduled again, for example. A delay, resulting from additional data transfer, however, would be very serious which has led to the decision that any additional transfer should be avoided in any case.

Nevertheless, the final results must be sent back to the server. It is assumed that an exemplary chunk generates a result of structure [key:value; key:value; ...]. Since the test case deals with hit-counting per month, twelve [key:value] pairs are generated. In the following, such a result is estimated with 100 bytes per chunk, producing a total of 667 * 100 $\approx$ 66700 bytes (65kB). This data packets are afterwards transmitted in about 69 ms each.

Operations Making general assumptions about the time that is needed for each function call is probably the most difficult task. Basically three methods are suitable to deal with this task. The first one is the direct measurement or profiling on a test machine. The drawback of this method is its concreteness. All obtained values only refer to one machine, with one possible hardware specification, at a given time, in a particular state. A second approach is the counting of elementary operations of the algorithm, dependent on the input size. Such elementary or basic operations can be, for example, assignments, comparisons or arithmetic operations. This method leads to a third approach which is the big O notation. All three methods are capable to make statements about the processing time.

In order to describe the runtime of the JSMapReduce framework, a mixture of the first and second approach is used, since an asymptotic notation such as big O would not be concrete enough. Based on simple tests, regarding the above described example case, a constant for arbitrary but sufficiently simple operations is assumed to be about **6,670 ms** per iteration over an array, containing all log file entries. This corresponds to approximately 6,670 ms / 6,669,204 lines = 0,001 ms per array element. A basic operation is thereby defined as an operation or function applied to one array element, that is map and reduce are seen as 'basic operations'. Such an operation has in turn different steps like comparisons or string manipulations which are not considered further, as long as they do not go beyond the scope of 'simple' calculations. This abstraction is useful to concentrate on the asymptotic behavior in response to increasing input sizes.

Conclusion While the runtime of a single-task program depends mostly on input data size and complexity of operations, the bottlenecks in the JSMapReduce framework, are much more complex and interdependent. Critical parts are for example data transmission, the number of connected workers, the performance of the server, its Internet access performance and worker failures.

In order to extend the current image of the framework, the following section will concentrate on gaining information in practice, that is, with experiments and tests.

5.1.2 Experimental Assessment

Reading and Writing First, a Java program is written which is aimed to measure the time, needed for reading the lines of an one gigabyte large log file, using a buffered reader. The measured time is 4.842 seconds.

In the next step, each line is not only read but also written into a copy. For this reason, the program is extended with a file writer. The processing requires 20.793 s minus the time for reading the file. The result is 15.951 s for the mere writing. At this point must be noted that the processing time varies depending on the number of processes and their IO operations.

Operations With the above analysis, the values for the runtime could not be estimated with sufficient accuracy. Nonetheless, the running time is likely to be the decisive criterion whether a flexible framework can prevail against task-related solutions. Thus, the next test is used to measure the performance of a comparable Java program. To measure the particular time for each operation, they are built and tested separately from each other.

The first measurement is done to determine the time, needed for decomposing the log file into its individual parts. These parts are, for example, the IP, date, requested page, and user agent. Next, the date part is further decomposed in order to extract the month. Five measurements are made at different daytimes. The mean of these measurements is 31.054 seconds. This value represents a difference of about **26.212 s** compared to to the mere reading of the file.

In a second step, each month is stored in a hash map. This step allows a simple incrementation for each newly evaluated month. Again, the program is executed multiple times at different daytimes to determine the mean. The runtime of the extended program is about 31.706 seconds, which is an additional delay of **0.652 seconds** for counting.

Worker Code The described steps are one possibility to solve the problem of analyzing a server log file and counting the number of accesses per month. The main loop for this evaluation is shown in listing 5.1. Even if the described

Step	Substep	Method	Est. Time
Data generation	-	-	-
Write data	Read data from file	hdparm / Java	[4.842]* / 4.740
	Partitioning	Constant	0
	Write to DB	hdparm / Java	11.962* / [15.951]
Read data	Read from DB	hdparm / Java	[4.842]* / 4.740
	Response generation	Const.	0
	HTTP transfer	Speed tests	1,118*
	Receiving data	Const.	0
Processing	Data structuring	Const./Java	[6.670]* / 8.989
	Map	Const./Java	[6.670]* / 8.989
	Group	Const./Java	[6.670]* / 8.989
	Reduce	Const./Java	[6.670]* / 8.989
	HTTP transfer	Speed tests	27.5*
Write results	Aggregate result	-	-
	Write to file	-	-
Total			1197,815

Table 5.2: Comparison II: Steps With Estimated Time[3]

solution may not be the best, it is written comparable to the JSMapReduce functions, allowing comparisons on a functional level.

```
String line;
String extractedMonth="";
String[] splitArray=null;
HashMap<String, Integer> months = new HashMap<String,Integer> ();
while ((line = in.readLine()) != null) {
  splitArray = line.split(" - ");
  splitArray = splitArray[1].split(" ");
  extractedMonth = splitArray[1].substring(1, splitArray[1].length
      ());
  int count = months.containsKey(extractedMonth) ? months.get(
      extractedMonth) : 0;
  months.put(extractedMonth, count + 1);
}
```

Listing 5.1: Java Worker

5.1.3 Putting Pieces Together

The preceding parts attempted to sketch the individual dimension of each processing step either by estimation or experimental tests. It was particularly important to define reference values in order to classify and rate later results.

Summing up all the partial results and the used methodology, the following comparison 5.2 is obtained.

The values for the read and write operations, on the test system, were determined in several tests with hdparm and afterwards extrapolated. In addition, an one GB large file was read and written with a Java program. Both values for the reading operation are almost equal.

In the following, it was pointed that the effort needed for writing additional metadata can be neglected. For this reason the data storage is stated with the simple write delay for 1 GB. Again, there are two values - one for the extrapolation and one for the Java measurement. Since the write speed varies with the number of disk accessing processes the greater value (15.951 s) of both is more likely to represent the benchmark.

Estimating a general, valid constant for the partitioning is virtually not possible, since the operations are different for each task. In the particular example task, the lines had to be stored separately from each other. While reading the file line by line, the partitioning is done implicitly which is why the preprocessing time is given by 0 seconds.

Reading from the database, has the same value as reading from a file. Writing to the output buffer with PHP echo was approximated with 0 s.

The transfer time, needed to supply a single worker sequentially with new data chunks, was extrapolated, based on the average transfer time. The transfer time, in turn, derived from the bandwidth statistics given by the speed test providers. With 1,118 seconds, the data transfer is the absolute bottleneck and more than 20 times higher than all the other operations in total. This has the consequence that the success of each job is dependent on how efficiently the data can be transferred.

Receiving the response in the browser is estimated at 0 seconds. The situation is different if the data has to be split again and stored into an array. According to previous tests, the constant for such operations was assumed to be 0.001 ms. With respect to this assumption, the time needed for 6,669,204 rows is 6.670 s. This value encompasses the splitting of the data, applying a function to each array element and grouping elements by the same keys. The time, needed for the reduce step, should be much lower, but for simplicity reasons it is estimated with the same value.

Although the individual processing (sub-)steps are different, compared to those in the Java program, the total processing time of the Java program is split equally onto the four main operations(see table 5.2).

Consequently, the total processing time for one worker would be 1197,815 seconds.

[3] All values refer to an one gigabyte sized log file. Values with '*' are extrapolated. All times are given in seconds.

5.2 Testing Platform: Snake

In order to obtain meaningful test results, it is important to have access to a consistently high number of workers. On the one hand, it is evident that a cluster with just one single worker is not and will not be competitive. But apart from that, it should be verified whether the assumptions about the web clients' performance and their cluster characteristics match the conditions.

To check these assumptions a test platform, that is, a special web site is build up and visited by Internet surfers. Computational jobs are distributed with the proposed JSMapReduce framework (see chapter 'Architecture' on site 20) to each web client that visits this prepared test page. The worker code is then executed on every client to process the allocated jobs while the individual performance of the clients is profiled in the background. The respective results are constantly sent back to the server.

The difficulties consist in finding enough volunteer testers to participate in such a preliminary test. One possibility to get volunteers is to rely on cooperation and to simply ask them to keep a tab or window open in their browsers for a certain amount of time. Another option is to make this type of test more intensive and interactive. One idea, in which the calculations are integrated into a browser game seemed particularly suited.

The following reasons illustrate the benefits of a game as test case:

- The online gaming industry is a growth market
- It is expected that the average playing time per Internet user will increase in the next years[43]
- Games often require no paging and thus 'guarantee' a certain time frame
- Error detection and load balancing are highly simplified by fixed time slices
- Games are fun - therefore, the test person does not feel used or tied to an open tab / window
- Volunteers are attracted, since they are (hopefully) interested in the game
- Games are computationally intensive and offer many scenarios where outsourced tasks can improve the total performance and game experience

Games enable a win-win situation where game provider and player can both benefit directly from one another. Moreover, does the computational intensity of the game offer two interesting opportunities.

- First, can be checked whether concurrent computations are possible at all. Concurrency problems were already outlined in the section on JavaScript. Now, it is checked how mature and advanced the concept of web workers was implemented by the browser vendors.

- The second aspect is how the performance of the browser is affected by high load and large amounts of data.

Through the combination of browser game and web worker, a comprehensive benchmark is constructed, testing unusually challenging situations in the browser. Both, the game and the worker process, consume massive resources and compete in their execution. If it is possible to play a JavaScript-based game smoothly, while still performing extensive computations in the background, this kind of proofs that the deployment of the described technology is feasible without limiting the user's surf experience at all.

Construction As suiting browser game is the mobile phone classic 'Snake'. The advantages are its well-known and simple gameplay as well as its low implementation costs. The field is drawn frequently, leading to high, but not too high, load on the resources. It should therefore be unlikely that the game causes unresponsive scripts on its own.

For the graphical representation of the field, the HTML canvas tag is used. To provide simpler access, a small library for creating primitives was written in JavaScript. The entire game logic as well as core functionalities such as collision detection or keyboard handling were also implemented in JavaScript. In particular was tried to keep all these components small and independent from external libraries. To create further incentive and to motivate the players, an additional AJAX driven high score was integrated.

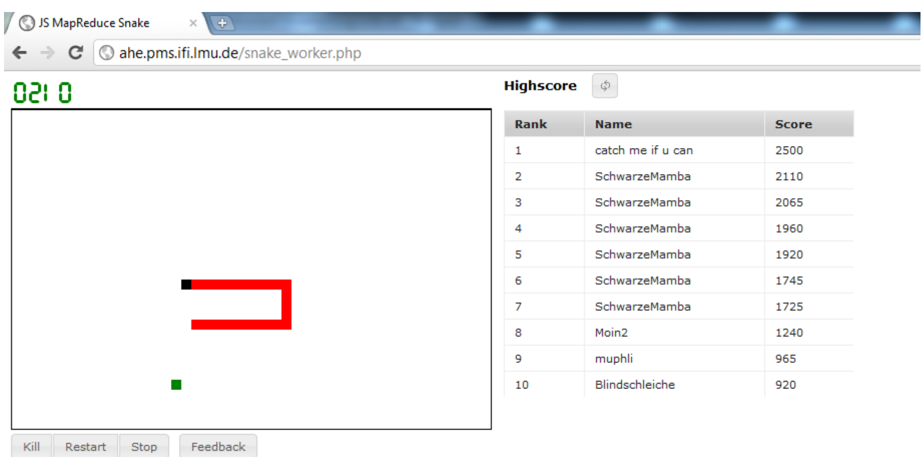


Figure 5.2: Snake UI

To illustrate the concept, the start routine and the main loop are displayed in the following listing.

```
var canvasAPI;
var world;
var keys;
var gameID;
function startGame(){

  canvasAPI = new CanvasAPI(480,320+30);
  var canvasTag = canvasAPI.init();
  document.body.insertBefore(canvasTag, document.body.firstChild)

  world = new World();
  world.init();

  keys = new KeyListener();
  keys.init();

  keys.registerHandler({
    DOWN: function(){world.snake.setMove("DOWN");},
    UP: function(){world.snake.setMove("UP");},
    LEFT: function(){world.snake.setMove("LEFT");},
    RIGHT: function(){world.snake.setMove("RIGHT");}
  });

  var fps = 30;
  gameID =setInterval(function() {
     update();
  }, 1000/fps);

}

function update(){
  world.update();
  world.draw();

  if(world.finished==true){
    clearInterval(gameID);
    finishGame();
  }
}
```

Listing 5.2: Snake Main Loop

As already described in previous parts of this work, it is not possible to ensure a comprehensive browser support for the game. For this reason, the visitor's browser first needs to be checked on compatibility. If the browser fits all criteria, the visitor is redirected to the game. Else, he gets informed that the used browser does not meet the requirements.

In addition, all players will be informed about possible stability risks and sorts of data storage. Only if a modal dialog is confirmed with 'OK', the main loop of the game starts. The player then sees an UI like the one shown in figure 5.2[4] and can play. Along with this, a new world-object is created in the background and the

[4]which is rendered with Chrome

key handler for controlling the snake's movements is initialized. The setInterval method (see listing 5.2 line 25) is used to frequently call the update method within the specified interval (see listing 5.2 line 24-27). The update method evaluates the length of the snake and its position as well as eaten fruits and the reached score. Afterwards, the field is redrawn. Once the game is finished, the setIntervall method needs to be terminated with clearInterval. Using the finishGame routine, the player is allowed to publish his high scores. But, before a player can publish his high scores, he is asked to fill out an evaluation form. The evaluation should be performed only once per player, which is why cookies need to be checked.

5.3 Evaluation

With the described testing platform, it was possible to observe and examine the system in live-mode. Even if the overall feedback from the players was positive, it revealed nothing about the actual performance and usability. Many assumptions and estimations were made in advance to the tests. The next subsections are aimed to analyze the received test results and obtain information with regard to the quality of such assumptions and for future application fields.

5.3.1 Overview

By analyzing the entries in the log files, it was possible to obtain a rough and global overview. The following values were determined on the basis of the application and server log files.

- 70,389 jobs were assigned.
- 70,219 jobs were completed representing a success rate of 99.75 %, respectively a failure rate of 0.25%.
- All in all, 107,836 MB of data was transmitted.
- 240 different workers were registered, leading to an average of 293 jobs processed by every worker.
- On average, each player processed about 449 MB of data in the background, while playing.
- The average playtime was about 4 minutes in the arithmetic mean. The median was about 2:11 minutes.
- Each player played an average of 3 games.

How accurate and meaningful these global values are, is examined in detail in the following distinction and consideration.

5.3.2 Worker Performance

To gain information about the worker performance, the measurement was done with basically two different approaches.

Approach I - General Performance This section describes the more general approach for measuring the worker performance. The measurement was done as follows:

1. Once a worker registers and starts the game, a first time point is set.
2. Starting at this point, the running time is measured until the stopping of the game which also induces the worker's stop.

Since the worker's runtime is coupled to the one of the game and a counter is incremented for each job that is processed, this approach measures the ratio between jobs and played time. From the runtime and the processed jobs, results the average work done per time - the **general worker performance**:

- The fastest worker completed 976 jobs in 199 seconds. This corresponds to an average processing time of about 204 ms per job.
- The slowest worker completed 39 jobs in 321 seconds and thus had an average processing time of 8.238 ms per job.
- The average processing time was 503 ms / job.
- The failure rate of the workers was 9.5%[5]

All results are limited by the type of measurement and the corresponding accuracy. One example is that prior to the tests a security setting was made that ensured that a worker does not request more jobs if a limit of more than 1,000 completed jobs is reached. On the one hand, it seemed unlikely that this limit could be achieved and plausible, since the original job consisted only of 667 chunks. On the other hand, it should prevent that a worker is not terminated after a fault and as a consequence floods the server with requests.

In fact, this limit was exceeded 14 times. Several workers also reached job counts far beyond 900. As positive as the result is, the security setting, nonetheless, prevented to determine the maximum of jobs that could be processed by one single worker. Even more unfortunate is the fact that the job processing was terminated, whereas the game was still running. For this reason, the time was not stopped with the result of extremely high processing times for such, actual, performant workers. As a consequence, the workers who processed more than 1,000 jobs were removed from the statistics in order to not distort it. The actual results were thereby adjusted downwards.

[5] This value should not be confused with the above-noted job failure rate, since a failing worker, normally, does not request further jobs and therefore not entail further errors.

Additionally, some workers had errors which caused them having significantly higher processing times. A total of 14 of them triggered the onerror event of the web worker API. Details about the appearing problem were not logged yet. In later tests such failures should be reported directly to the server in order to handle them. Moreover, it is possible that unnoticed problems occurred besides those detected.

The measured times of incorrect workers were left out from the calculation of the average processing time too, since, once again, the time was stopped although the workers had little to no longer worked.

A third exception are eight workers who were excluded from the statistics, as they have not processed any job. All eight workers have in common that their played time is below one second. The reason is apparently that a player has restarted in rapid succession.

Excluding the three above-mentioned special cases the results, displayed in figure 5.3 and 5.4, were obtained.

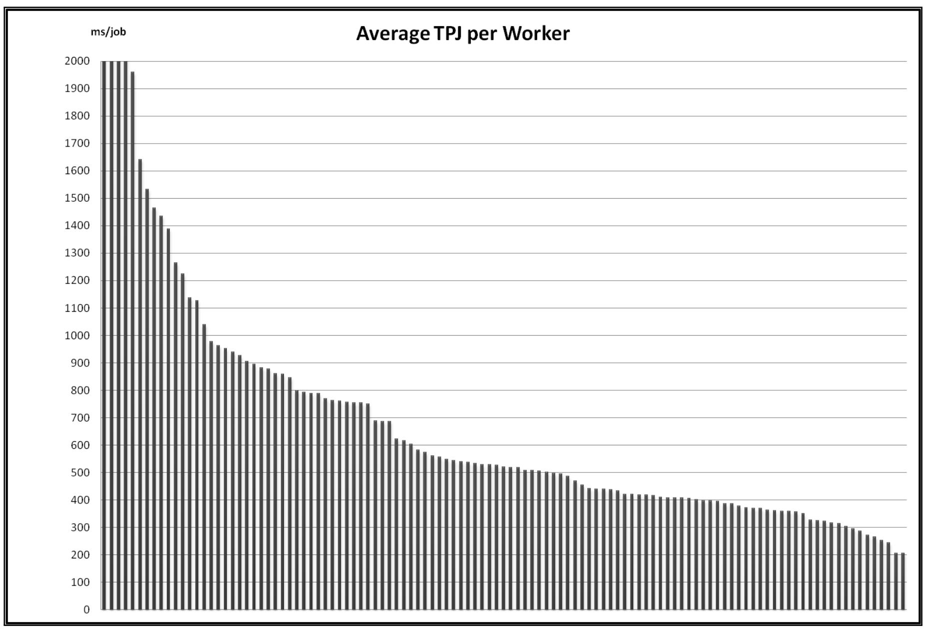


Figure 5.3: Average TPJ per Worker

The results from 5.3 and 5.4 include all workers with unique worker number. This means, if a player starts the game again, it is possible that the same machine with its individual performance is included several times in the statistics. Since there is no machine which has an extraordinary high number of registrations and local networks can have one IP for multiple machines, this fact has been neglected.

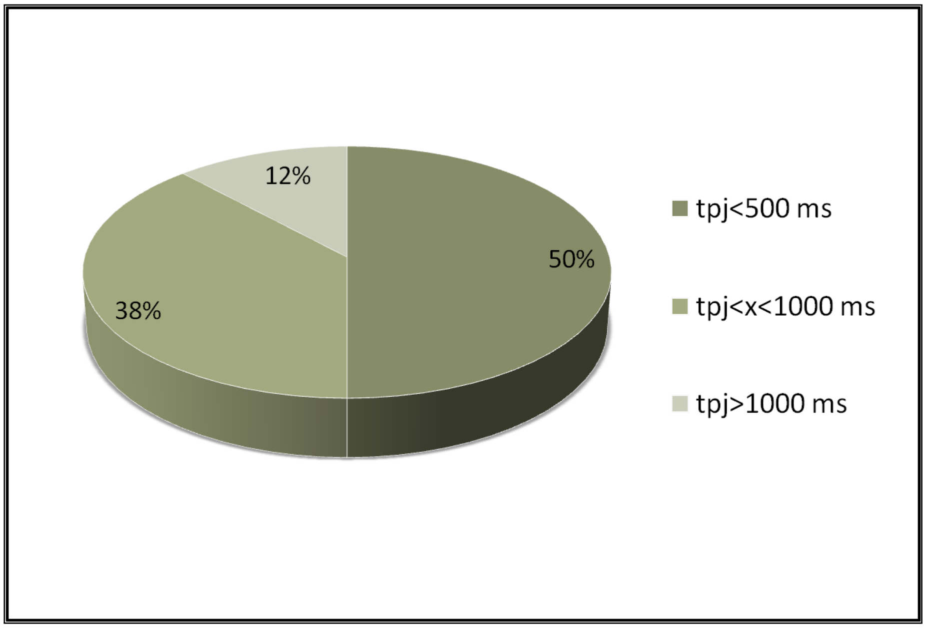


Figure 5.4: Performance Cohorts

Figure 5.3 illustrates the average Time per Job (TPJ) per Worker, i.e. each bar on the x-axis represents one worker with its individual, average performance.

Figure 5.4 shows groups or cohorts of workers that share the same performance class.

Using the mean value, the results can be divided evenly into two groups. Fast workers needed for requesting and calculating a job less than 500 ms, whereas slower workers needed between 500 ms and 1000 ms. Additionally, there were some very slow workers which needed more than 1000 ms per job. One interesting thing is that 11 of these 15 very slow workers had a playtime less than 4 seconds. Obviously, the game was stopped in a very early stage, leading to a massive delay and a low TPJ for the first jobs, thus, a weak overall performance. Values that result from such effects, should be rather counted to the failure rate than included into the performance statistics. These values represent, in a sense, the bounce rate, that is visitors who leave the site very quickly.

Apart from these bounced, slow workers, there were still another four workers that calculated only a few jobs, although they had higher playtimes. Reasons for this may be failures, the used hardware, the computational performance or load on the PC or browser. Table 5.3 provides additional information to these outliers.

Time per Job (TPJ) [ms/job]	User Agent Info
8238.51	Windows XP; Firefox/6.0.2
3753	Windows XP; Firefox/5.0
3630.36	Mac OS X 10.6; Firefox/6.0.2
2811.27	Windows 7; Firefox/3.6.20

Table 5.3: Slowest Workers

As can be seen, two out of four of these slow workers had both, an outdated OS and browser. Further, can be assumed that the hardware of both was not contemporary as well. While one worker has indeed a current OS, its browser version is completely out of date. Even if this browser version is listed as web worker compatible, it is likely that it was not supported very well and maybe caused errors. The last combination is Mac which has an recent system but an older browser version.

In contrast, the field of the fastest workers is quite heterogeneous. There are no irregularities with respect to the used software. The best performers include, for example, one machine running Ubuntu and Firefox 8, but also a Mac with Safari or a computer with Windows 7 (64bit) and Chrome in version 15.

Finally, those machines are inspected which had signaled a fatal error, that is, the onerror event of the web worker API was triggered.

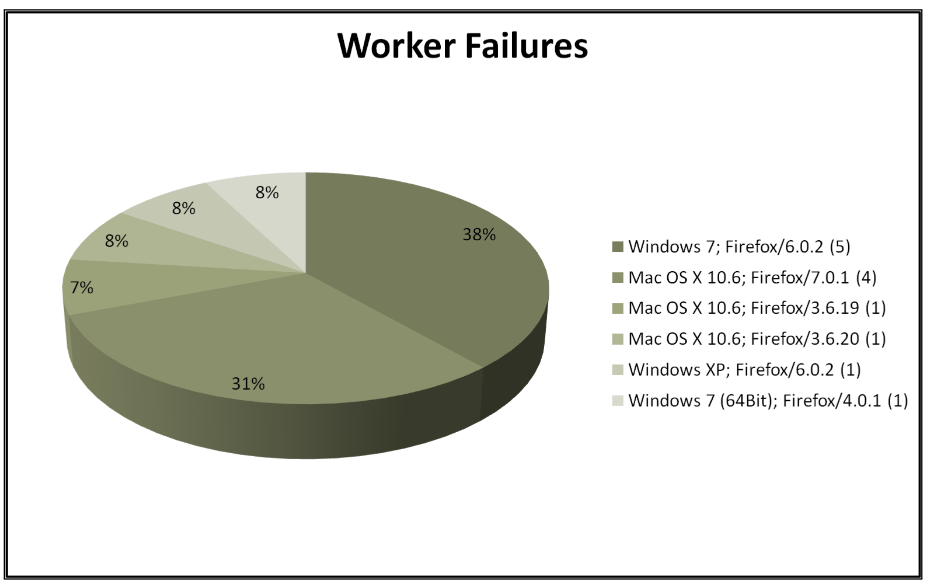


Figure 5.5: Failed Workers

Each of these workers met the requirements to access the web worker API. Again, it is striking that, with exception of one, all workers used either an outdated browser or operating system. Very striking is that one particular combination of Mac OS X with Firefox 7 seems prone to errors although, both, the system and the browser should fit the requirements. But even the machines running Windows 7 and Firefox 6, should not have come to a noticeable error rate.

Further tests could check whether this is an implementation-specific problem of the particular browser version by building clusters of similar configured machines. With the described method the average and individual worker performance could be determined and examined in relation to either occurred errors or the used operating system. Through further analysis, error-prone systems, browser versions and browser-OS-combinations could be detected. Nevertheless, it has already been demonstrated that an outdated browser or OS is a risk for the system's stability.

With this first approach, data was collected from 148 of 240 workers which is just about 62%. The reason for this is that data was only transferred with the end of the game and not prospectively. With the following, more detailed approach, it was possible to obtain more data and with higher granularity by transferring results continuously.

Approach II - Main Loop The previous values provided an overview or the 'big picture', that is, the totally achieved performance. Further, did some bottlenecks and points at which workers fail with high probability became apparent. The following approach deals with a more detailed analysis by profiling several computational steps.

In optimization terms, the characteristics of the web clients can not be changed, but however the architecture and implementation of the worker code. For this reason, one additional measurement was made such that the processing times for the relevant steps, as they are described in previous sections, are captured as well. This data is used to provide information about critical and interesting program parts and to detect power peaks, performance fluctuations, and malfunctions.

For each client and with every computation, following profiling information was recorded.

- Request: Time between the request of a job and the arrival of the associated data.
- Initiation: Decompressing, parsing and Array initialization.
- Map: The Time, needed for applying a custom map function with the higher order map on every record in a dataset.
- Group: Grouping together all records which share the same key.

- Reduce: The Time, needed for applying a custom reduce function with the higher order map on every record in a dataset.

The profiling resulted in a total of 70,215 measurements in 70,219 jobs.

Request Time The request time has been identified as the most critical value in the whole framework and overall bottleneck. It represents and includes both the transfer time or Round Trip Time (RTT) and the server-side processing of a job request. The server-side processing, in turn, is made of the searching for unassigned jobs, the determination of associated data chunks and the reading of these chunks. The respective measured values for the individual request times are presented in the diagram 5.6.

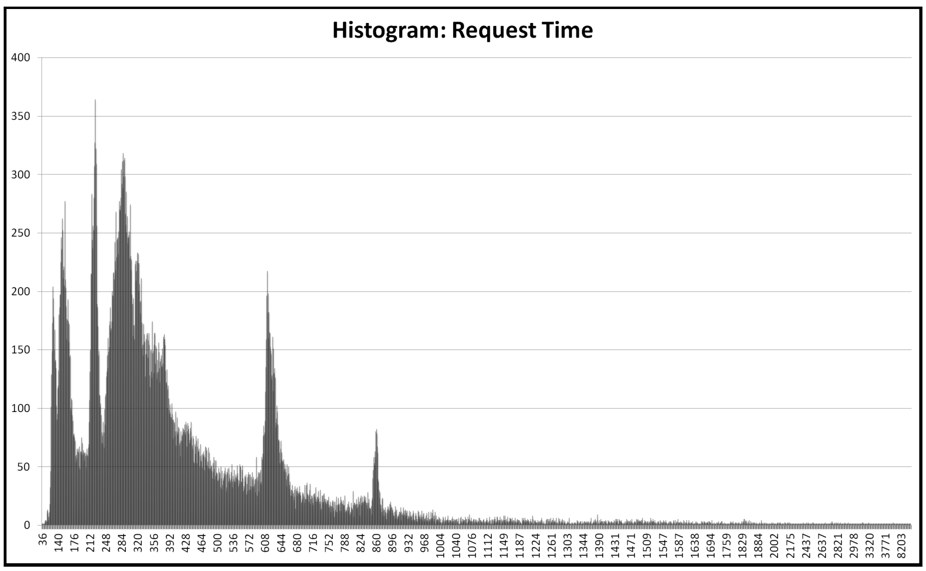


Figure 5.6: Histogram - Request Time

The histogram 5.6 illustrates the value distribution by assigning the occurred request times to the x-axis and the corresponding count, each value occurred with, to the y-axis.

The peaks in the diagram further represent the broadband rates of the German broadband providers:

'DSL 6000' which is the most common tariff, for example, enables transfer rates up to 5,008 kbit/s[7]. A data packet of size 1.532 MB ($\approx$ 12.851kbit), would cost an expensive transmission time of about 2.56 s. This value, however, differs significantly from all plotted peaks. The reason for this is the used gzip compression between server and client. The compression of data can be illustrated well by capturing the TCP stream with a tool like Wireshark. But even simpler browser extensions like Firebug or 'HTTP Headers' are able to show captured header information and other useful hints.

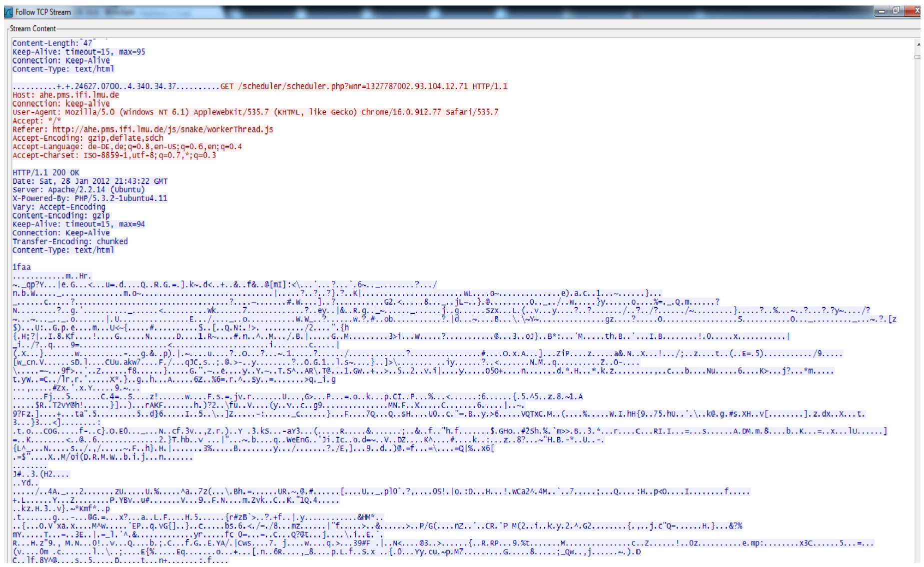


Figure 5.7: Wireshark: Compression

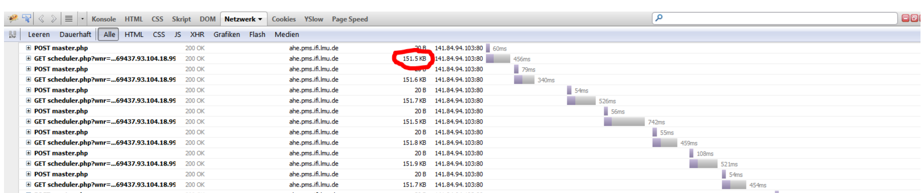


Figure 5.8: Firebug: Network Panel

The Wireshark capturing takes place just before the browser decompresses the received data. Therefore, the data is not human-readable although the lines are from a standard ASCII log file. Additionally, client and server have to settle the compression causing the respective entries in the header fields, which can also be seen in both figures. Browser extensions like the mentioned Firebug access the data after decompression resulting in a contradiction between listed package size and the actual decompressed size (see fig. 5.8).

On closer examination of the HTTP headers in Firebug, one can see that the data is transferred compressed and with a size of $\approx$ 151.5 kB, ie, $\approx$ 1241 kbit and a corresponding compression rate of about 90.3%. With an assumed transfer rate of 5008 kbit/s for the dsl 6000 tariff, the data could be transmitted in nearly 0.248ms.

	Request	Initiation	Map	Group	Reduce	Total
Mean	423.36	21.89	26.47	3.70	0.02	475.44
Median	328	18	19	3	0	368
Mean (corrected)	356.20	18.17	20.32	3.33	0	398.02
Median (corrected)	316	17	18	3	0	354

Table 5.4: Profiling: Average Runtime

The next peak at about 600 ms corresponds to a doubling of the transmission time, that is, a halving of transmission speed. This result also reflects the tariff policy of the broadband providers.

In the front area, the boundaries of the individual peaks, are more and more blurred, which can be reasoned best with the fact that even in the next higher rate class the full throughput is neither guaranteed nor reached permanently. The effective achieved transfer rate depends for example on the location or network load.

The very first values in the front area are particularly striking. A transfer time of 32ms seems very questionable, considering usual connections and their respective transfer times. The most plausible explanation is that the multiple of a fixed chunk size does not always match the actual file size. For this reason, there are often a couple of lines that remain and build the last chunk. The low occurrence of such exceptions underscores their small peak.

The outstanding gzip compression of text files is clearly reflected in the results. While having estimated 2 seconds per chunk, the measurements show that due to the implicit compression many chunks actually need only less than 200 ms. Unfortunately, the 90% compression rate and the respective transmission time already marks the lower bound for transfering such 'text chunks'.

MapReduce Once the data has been transferred to the clients, the actual calculation begins. While the request time says less about the performance of the machines but on their connection and the server, the profiling of the respective function calls determines the actual achievable worker performance.

Table 5.4 lists the obtained average values for the runtime of each time-consuming, computational step. Since the average is strongly affected by faulty workers and extreme values, a great difference between the arithmetic mean and the median results. For this reason it is useful to leave out all values related to workers that would be excluded from a real job as well. The threshold values are 800 ms, 40 ms, 40 ms and 8 ms in their in table 5.4 used order. With this adjustment, the corrected values are obtained.

Correlation The mean and median values of the respective processing times are characterizing an expectation value for the overall achievable performance.

In addition, these average values yield a lower and upper bound.

Since the computation time for each step differs from client to client and even within the session of a single client, further analysis is needed. The strength of the intra- and inter-client performance fluctuations is a criterion for efficient scheduling and distribution strategies as well as fault tolerance. Moreover, it is decisive for calculating a stable aggregated computing power and does not emerge from the simple monitoring of mean values.

Because the total processing time can be massively delayed by slow or partial slow calculations, it is important to detect and handle slow workers. First of all, this includes the re-execution of the respective assigned jobs. More important, however, is to consider the exclusion of such workers from further processing. Some approaches, dealing with this, have already been discussed in preceding parts. The following parts will try, however, to examine what criteria exist to exclude a worker partial, that is, from one particular task or general, thus not only from the actual but all following jobs.

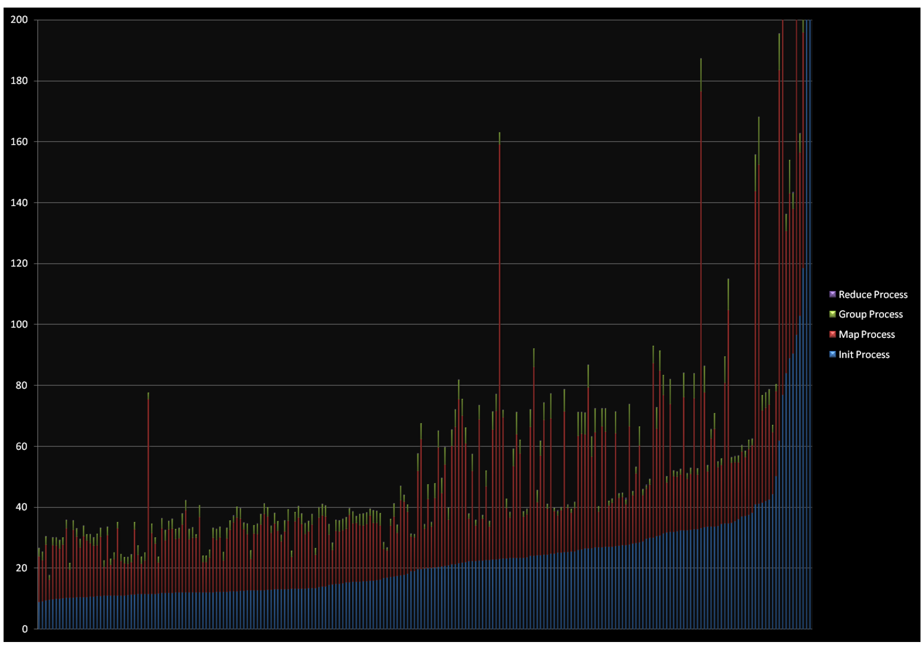


Figure 5.9: Profiling: MapReduce

The chart 5.9 illustrates the individual total processing times with one bar for each worker. Each bar, in turn, consists of the cumulated mean values for the main, computational steps 'initiation', 'map', 'group' and 'reduce'. The bars are, in particular, sorted in ascending order by their average init time. This representation allows to analyze the interacting of the individual processing steps.

As an initial hypothesis, one could suppose that a worker that completes one partial step very quickly, in general, processes other steps in a similar way. In fact, chart 5.9 shows already significant exceptions to this assumption.

First, a classification is made. According to table 5.4 of the previous section, workers with an init time less than 20ms are defined as above-average or fast. Moreover, all the workers with a processing time of 20-40ms are average performant and last, those workers with a processing time of more than 40ms are defined as too slow which is equal to be down or crashed. Therefore, such workers should not complete the job but release it for others.

Considering the class of the workers that processed the initial sub steps the fastest, at least two facts are striking. The one thing is that the initial hypothesis can be largely confirmed for this class which means that a worker who was above-average in the first step is above-average in all steps. This can be proofed from the fact that the total processing time is almost without exceptions less than 40ms for the entire field.
The second conspicuousness is that there are still considerable fluctuations in the computation time for the map step in this group. Contrary to the assumption that the worker who calculated init the fastest is the fastest for the map computation too, there is no consistent increase for the map values.

Even more apparent become these irregularities in the field of the average workers. Within this group, the computational time for a slow map calculation is often double or even triple times higher compared to a fast calculation. Although many workers needed less than 20ms for the map which is a good result, this group contains significantly more workers that processed the map in an unreasonably time. It can also be noted that those workers who had a bad processing time for the map step are very similar in their total processing time (about 70ms). One explanation for this could be that operations have been implemented differently in different browsers.

The slower the init time of the workers is, the more frequent are calculation times beyond 80ms and more. Total processing times, reaching a limit of above 100ms, are in the final analysis comparable to malfunctions. Nonetheless, it is surprising that even within these very slow workers some map calculations exist that are faster than the average.

Based on this representation, the question of how to exclude a worker partial or general must be asked again. In order to assess an individual performance, it is important to determine a mean or median as pivot element for the runtimes of the first operation. According to the preceding analysis, it then may be possible to recognize performance classes in relation to this value. In a next step, all slow workers could be excluded, since the probability to stay slow increases strongly for those with a slow start. This strategy, however, would be associated with the risk of loosing many potentially good workers which had only a slow start. At the same time a lot of computational power would be lost while it is unclear whether the initial fast workers stay fast. An important piece of information to make that decision and thereby minimize the risk is whether and how much a single worker varies in its performance within its total life time.

In order to provide this information, the values of all workers, first, are grouped and sorted in ascending order by their average init times again. Subsequently, the differences between the maximum and minimum processing time of init and map are calculated. Since init and map have the highest impact on the total processing time, the resulting chart can be used to determine inter procedural fluctuations.

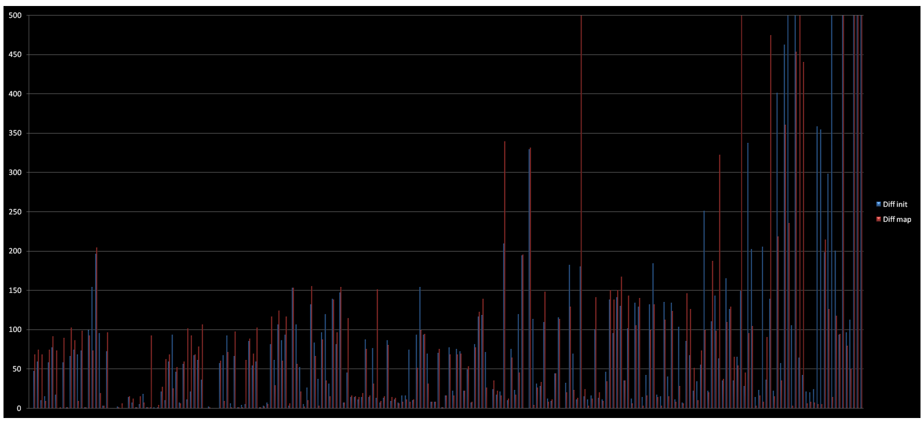


Figure 5.10: Intra Client Performance Fluctuation

Figure 5.10 shows this maximum measured fluctuation within workers for both the init (blue) and map (red) computation. Due to their same order, figure 5.10 is compatible to figure 5.9. The fluctuations refer in each case to one single worker. In the case of a small fluctuation, a fast worker (mainly in the front area) is consistently faster than average, at the same time, a slow worker with little fluctuation is consistently bad.

As illustrated with figure 5.9, the performance of a worker does vary, depending on the actual step that is processed. According to figure 5.10 are workers with fast init computation dominated by fluctuations for the map processing whereas workers with slow init times have much higher blue peaks, standing for a high init fluctuation.

Another striking point is that the fluctuations are significantly higher for slow workers. The highest bars were cut off at a limit of 500 ms. This implies that even a bad worker, can for example calculate 90% of its time with an above-average performance and still be slow in the end result if only one calculation was a total failure. The overall high values suggest that it would be wrong in terms of a distribution strategy to exclude such a worker completely or generally just because of, e.g., a bad start. The right decision, however, would be to exclude such a worker partial, that is, only for a certain time frame or number

	Initiation	Map
Mean	122,56	204,83
Median	52	37
Mean (corrected)	59,11	53,90
Median (corrected)	44	32

Table 5.5: Profiling: Average Fluctuation

of computations. The opposite is true for clients who are constantly working slowly, that is, without large fluctuations. The problem with this criterion, however, is that a worker, to be consistently bad, must have processed several jobs which is not conducive to the total performance at all.

Table 5.5 summarizes the average fluctuations. The average processing time for init is about 50 ms and 40 ms for map. Once more, it is notable that the average is inexpedient to specify a reference value, as it is too sensitive to extreme values. The limit for the corrected values, was at 300 ms, which is already a very tolerant time frame. Nevertheless, the average values remain very high. Comparing the average fluctuation of 40-50 ms with the average computation time, it does not seem to be unusual for a worker to have, now and then, computations with a duration that is three times as high as normal. Up to now, one parameter, however, has completely been neglected. In the following, the number of calculations is also included into considerations. Since the worker performance varies widely across the clients and an initially under-performing worker is not necessarily bad in the long term, it is the more important to know how many bad runs good workers may have. In this context, a 'bad job' is defined as a computation in which a part of the calculation needed more than the double average processing time. According to table 5.4, for both, init and map, the average processing time is assumed to be about 20 ms. Thus, a worker has failed if one of the computational steps takes longer than 40ms.

In order to determine an average value for the number of bad runs, made by good workers, all workers with an average total processing time of more than 40 ms were removed from the statistics first. Afterwards, calculations which are bad, according to the above definition, were counted for the remaining workers. This led to the following results:

- The cumulative overall probability of having a bad job was 1.35% for the group of good workers.
- 155/227 i.e. 68% of all workers exceeded such a timeout.
- 77% of them exceeded both, the init and map timeout.
- Both limits were exceeded nearly equal in their total.

In this last section, the performance of the workers was analyzed. It was shown, once again, that the difference between a slow and fast calculating worker can be significant. In addition, it was shown that even function calls of a single function can vary greatly in their execution time within one worker. Overall, it was illustrated that there are high differences between clients as well as function calls, characterizing the performance vagueness in clusters of web clients.

Contrary to this vagueness is the discovery of performance classes. It could be shown that workers which processed the first step in the test calculations very fast had high probabilities to come to good results for following calculations too. However, workers that have been slow in their first calculation only had a probability of about 30% to be over average in total. If slow workers could be detected at an early stage which in turn would prevent them from blocking jobs for fast workers, the overall performance would increase significantly.

The probability of having an irregularly high delay in the execution, was at 1.35% for fast workers.

5.3.3 Server Performance

The server performance has been largely neglected in the early tests, since failures were assumed to occur client-side. Further tests on the server, such as profiling MySQL queries, would have slowed down the entire system, resulting in longer loading delays and thereby being not helpful at all. In the next tests, the server and its code execution are analyzed, since the server-side processes do have a great impact on the overall performance. The used tools and techniques are based on a presentation by Derick Rethans[33].

A crucial question that must be answered before a system is modified is whether the system has to be optimized at all. Replacing arbitrary syntactical statements to gain little extra performance leads to more errors in most cases. So the first thing to check is whether the system is generally weak and how it behaves under load. Within this early tests, it must be taken into account that the requirements on the server change over time. Testing only the most realistic scenarios, is not sufficient enough in most cases and is not helpful in detecting bottlenecks. Instead, conditions should be tested that are forward looking, like, for example, increasing traffic rates.

Stress Test The first thing checked was how the system behaves under high load. Since it is not so easy to find enough volunteers for such tests, a tool called 'siege' did the job of generating requests and simulating visitors.

In order to authentically reflect the server's reaction on a great number of incoming requests, the tests were repeated several times. Each test lasted 30 seconds while the number of concurrent connected clients was doubled with every run. At the same time, the number of transferred jobs as well as the needed time were measured.

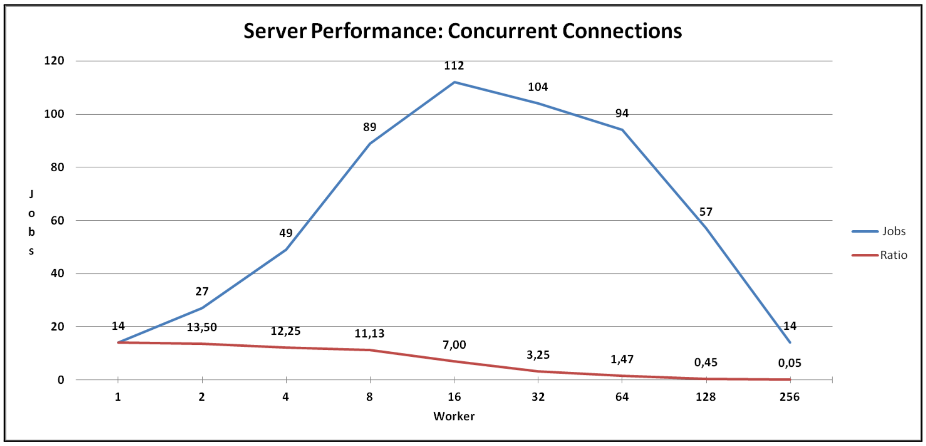


Figure 5.11: Server Performance: Concurrent Connections

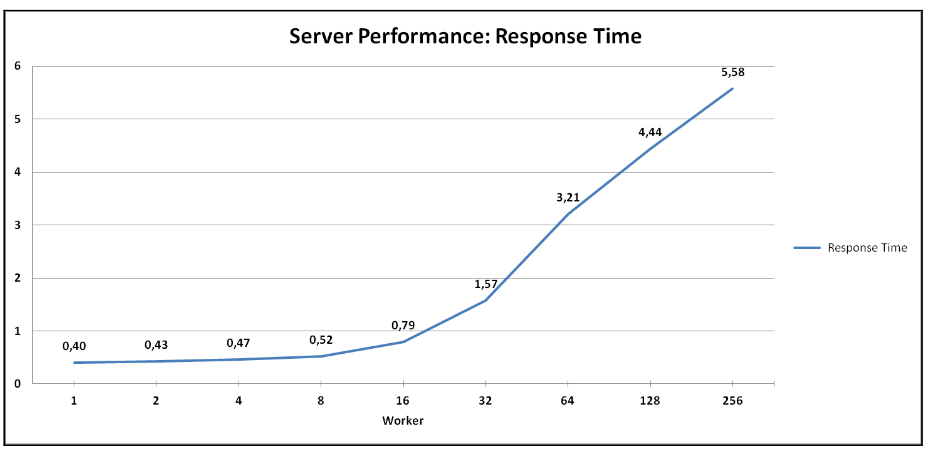


Figure 5.12: Server Performance: Response Time

It turned out that an estimated proportional increase in job-transfers is significantly limited. The number of transferred jobs is only (roughly) doubled with every doubling of the worker number to a limit of 8 concurrent clients (see 5.11). For more workers, however, the transfer count increases less and at the expense of stability. This congestion is amplified with every additional worker and finally it is irrelevant whether 256 or only one single worker is connected to the server since in both cases only 14 jobs are transferred within 30 seconds.

While the server is under stress it is useful to display some additional system-specific information, such as the used memory, I/O processes or CPU peaks. Linux does this for example with a tool called vmstat.

```
procs -----------memory-------- --system-- ----cpu----
 r  b free     buff  cache       in   cs   us  sy  id   wa
 0  0 11267668  36740 457252     100  165  0   0   100   0
 0  0 11267668  36740 457252     118  168  0   0   100   0
10  0 11267684  36740 457252     282  456  14  0   86    0
10  0 11267420  36748 457252     675  503  100 0   0     0
 2  0 11222504  36748 457252     1111 2329 55  4   41    0
 2  0 11245016  36748 457256     735  404  38  2   60    0
 6  0 11237492  36748 459012     768  748  72  2   26    0
 8  0 11228904  36748 459052     1078 1321 84  2   14    0
 9  0 11240900  36756 460828     982  992  95  2   3     0
 5  0 11224968  36756 460848     942 1373  96  2   2     0
 6  0 11200012  36772 469564     1077 1949 83  4   13    0
 2  0 11237292  36772 471360     741  923  59  2   39    0
 1  0 11237636  36780 473148     212  368  3   1   96    0
 0  0 11239876  36780 473152     164  175  0   0   100   0
```

Listing 5.3: vmstat Report

vmstat lists information on the memory, the system and the CPU. Whenever one of these components is too busy, the whole application runs in danger to crash. In other words, the database may performs too many simultaneous queries, there may be too many read and write operations or the code is just poorly written - vmstat provides more concrete evidence on what happens. The report shown in listing 5.3 was created while the system was under stress, due to siege. As can be seen in table 5.3, the value for 'us', that is, the user time increases rapidly until the system has little run-time. The remaining values are however fairly stable. Waiting periods for I/O, for example, do not take place at all. The results reflect on the one hand the mediocre performance of the virtual machine, but on the other hand indicate that code optimizations could improve the framework.

Concurrent Clients and Damping In order to predict and describe the system's behavior under load, the correlation between transferred jobs and the number of concurrent connected workers can be specified through regression analysis. The resulting approximated function can then be used to characterize the server's performance loss, caused by many concurrent clients and having an impact on the actual achieved distribution rate.

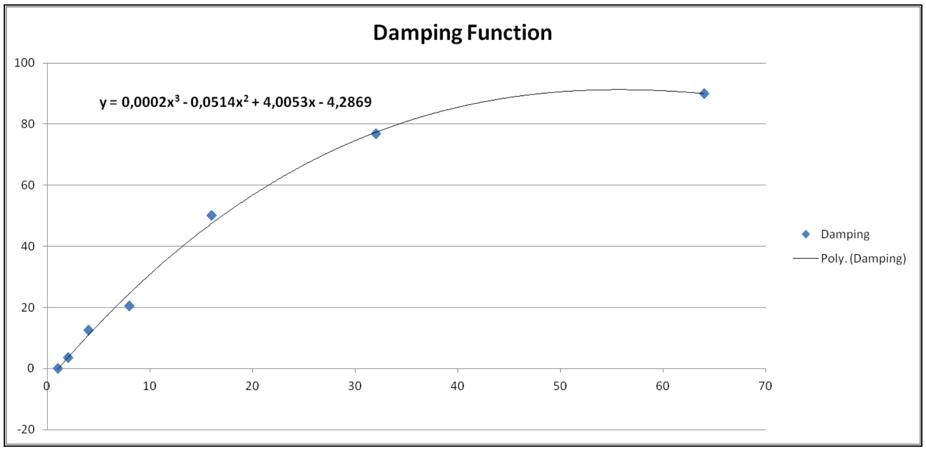


Figure 5.13: Damping Function

To obtain a magnitude for this 'damping', the ratio between the actually measured and ideal values is calculated. Following, these values are approximated with a function. Since the function is only an approximation its value range is limited. The resulting function is displayed in figure 5.13. The values on the x-axis represent the number of concurrent workers while the values on the y-axis illustrate the percentage of performance that is lost (shared). The values can be read as follows: While the server can provide a single worker with 100% of its expected power (14 jobs), it processes less than 50% of the expected requests for client numbers above 20 (jobs <140).

Code Profiling While the behavior of the system can be described with the above function, it is, however, interesting to find potential bottlenecks in the code in order to fix them. For that reason, the code is specifically examined as follow.

To obtain a simple overview, several timers were placed in the code and their output was passed to the client via firephp. The same applies to the client-side code where the output was printed to the Firebug console as well, resulting in a more comprehensive picture.

```
    $start = microtime(true);
    $data = getChunk($chunk);
```

```php
    $stop = microtime(true);
    $firephp->log(round(($stop-$start)*1000),'getChunk');
```

Listing 5.4: PHP Profiling: Timers

In order to get more detailed information on the system and the function calls, beside these profiling times, an additional PHP debugger such as is well suited. After the installation, it can be configured through the 'php.ini'. xdebug has several extensions regarding basic PHP functions, control mechanisms and modes. To obtain deep insights into function calls, xdebug must be put into profiling mode. In this mode, statistical, program related information can be captured. The captured data is written into log files that are not human-readable which is why they have to be analyzed further by additional applications such as kcachegrind.

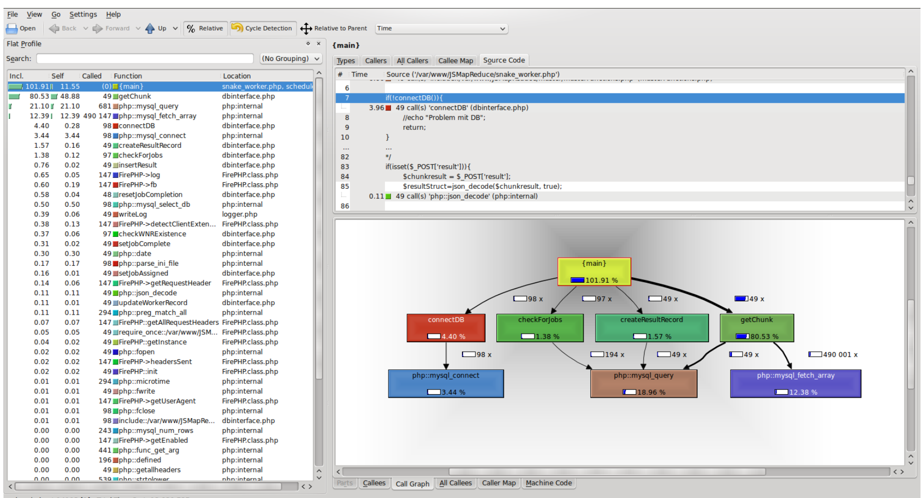


Figure 5.14: Server Profiling

kcachegrind processes the data from the log files and provides access to information such as each function's share in the total runtime. In addition, it can create a call graph with adjustable graph depth. In this call graph each weighted edge represents one or more function calls while each node represents a function. In addition to the call graph, the source code can be displayed, browsed and enriched with information from the profiling.

The data in the exemplary report 5.14, was obtained from a scheduler analysis. The report shows that the scheduler was invoked 97 times, that is, 97 jobs were distributed during the test. Moreover, the 'getChunk' method was responsible for 80.53% of the total runtime. This insight is valuable for the optimization, since it does not make sense to optimize functions that have little to no impact on processing delays.

The average run time of the scheduler script for one chunk is about 147 ms according to the xdebug logs and the manually set timers. One reason for this is that the virtual machine shares the hardware with other guest machines and the host's power is shared among them.

5.3.4 User Feedback

In addition to the pure performance, the user's opinion and assessment were evaluated as well. A survey pop-up was shown to the players at the end of the game, asking for the completion of two sentences. Both sentences had to be completed by moving a slider to one of the predefined answers. In total, there were 6 options per answer and a neutral slider position that was not accepted as final answer. Once a player evaluated the game, he was allowed to publish his high score and was not asked to evaluate the game anymore. 47 players completed this survey.

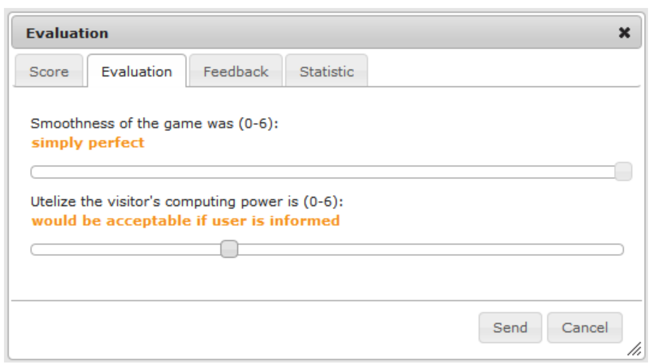


Figure 5.15: User Feedback

The first step was to gather information about the smoothness of the game, that is, the concurrency. The mere fact that calculations took place and were calculated reasonably performant reveals nothing about how the competing processes, game and worker, affected each other. A player, however, is able to perceive whether something bothers or is not functioning correctly, based on the game flow. The results of this survey are shown below in chart 5.16.

According to the evaluation, there was no player who could not play or had serious problems with the game. This in itself is a considerable success and demonstrates that compute-intensive, concurrent worker-processes can be realized with JavaScript. The effectiveness is confirmed even further by the fact that about 79% classified the game as well playable. Since a game is much more computationally intensive than any JavaScript code that is embedded in a normal website, the evaluation implies that similar worker threads can be

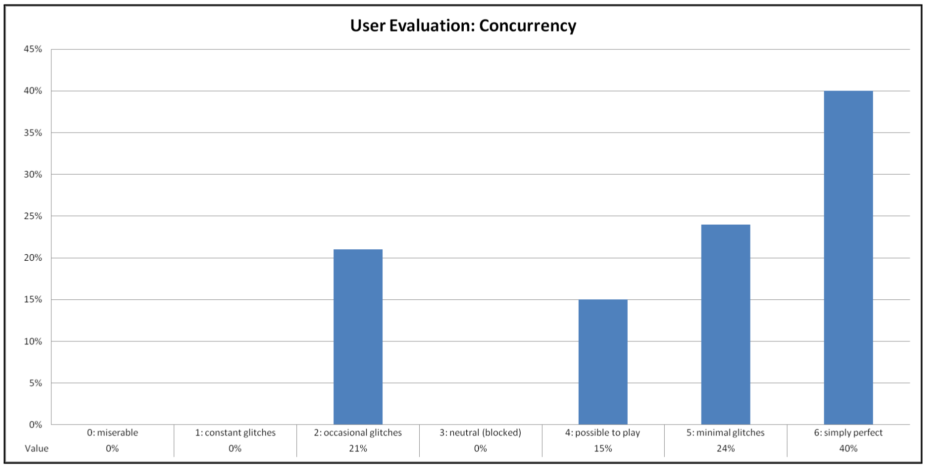


Figure 5.16: User Evaluation: Concurrency

used on nearly every website without affecting or restricting the responsivness or functioning.

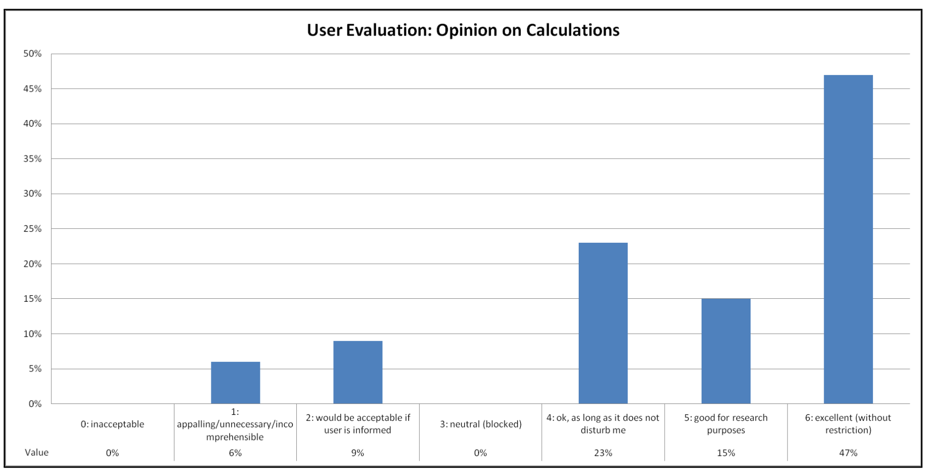


Figure 5.17: User Evaluation: Opinion

The second question was aimed at the player's personal opinion with respect to the sharing of his own computational power. Overall, the project was well received and the large part of the participants (75%) even endorsed the use of such scripts. Some attached the condition of being at least informed about the respective purpose or being asked by such scripts for permission to the deployment.

	Original File	Chunk	Compressed Chunk
Lines	6,669,204	10,000	10,000
Data Size	1 GB	1.532 MB	151.5 kB

Table 5.6: Comparison: Chunksizes

5.4 Comparison

Section 5.1 (Estimation) dealt with estimation techniques and was summarized with table 5.2 (page 44), listing all obtained values. In order to enable a final assessment, this table is now extended with values from the tests, as well as, values from a Java program.

The average processing times for the different computation steps were determined for each worker with approach II in section 5.3.2 and summed up in table 5.4. Since the measured values refer to (partially compressed) chunks, they must be adjusted, to be comparable with the estimated values. Table 5.6 provides an overview over the different sizes.

As already mentioned, the whole job consisted of 667 equal[6] chunks. The measured values in table 5.2 refer to uncompressed chunks, which is why they are remarkably larger. The average is represented with the median, because the arithmetic mean was to sensitive to outliers. These averages were extrapolated by being multiplied by 667. Thus, init and map which both had an average of 18 ms, were extrapolated to 12,006 ms.

Within the calculation of the transfer time, it was not taken into account that data is transmitted in compressed form. For this reason, the estimated values had to be corrected. Instead of 1 GB of data, only 667 * 151.5 kB = 101050.5 kB were effectively transferred. Accordingly, the estimated transmission time in table 5.7 was reduced to 107.787 s.

Moreover, the measured values for the transmission time do implicitly encompass server processing time. This can be explained as follows: If a timer is set in the client-side code at the beginning of a job request, the subsequent request process can be divided into several life cycle phases. Two important phases are, for example, 'waiting' and 'receiving'. In the wait state, the client is doing 'nothing' else but waiting. That means, if the request is processed asynchronous, the client's script is executed further and the client is informed about the end of the request, whereas a synchronous request would block the entire script until the arrival of the server response. Since AJAX uses typically an asynchronous communication, the request is non-blocking and the client's computations are not stopped. Meanwhile, the request is sent to the server, received and processed. After a certain time frame, which is determined by the RTT and the server's processing time, the wait state is terminated by an incoming response header,

[6] simplified assumption

Step	Substeps	Estimated	JSMR (single worker)	Java
Data generation	-	-	-	-
Preprocessing	File Read	4.842	4.740	-
	Partitioning	0	0.923	-
	Write to DB	15.951	21.265	-
Fetching Data	Requesting Data	0	n/a	-
	Server Processing	4.842	98.049	4.740
	Header Transfer	0	n/a	-
	Data Transfer	~~1,118~~ 107.787	112.723	-
			210.772	
Processing	Data structuring	6.670	12.006	26.212
	Map	6.670	12.006	-
	Group	6.670	2.001	0.652
	Reduce	6.670	0	-
	~~HTTP transfer~~	~~27.5~~	~~≈ 33.35~~	-
Write results	Aggregate result	n/a	n/a	-
	Write to file	-	-	-
Total		160.102	263.713	31.604

Table 5.7: Comparison III: Summary

that induces the receiving phase. In this communication stage, the server sends the requested data packets as TCP stream back to the client. The client in turn receives the data packets and builds an associated data object. In between, the data may be decompressed and prepared. Only if all steps have been processed completely, the client is able to work with the received data. In this case, the XHR object triggers the onreadystatechange event with readyState 4.

The individual request phases are difficult to distinguish from each other and are implemented differently in different browsers. The time, the client spent waiting, that is, from its request until the moment the data was ready to work with, could be measured accurately within the worker process. This time, however, should not be confused with the waiting phase in the HTTP communication process, which can be displayed by browser extensions like 'Firebug' or 'Chrome Developer Tools' in their network panel.

Another value that could be determined relatively accurately is the server processing time in the scheduler script. If this value is subtracted from the waiting time, the approximate RTT for request and response is obtained.[7]
According to this calculation it can be said that the runtime for the script was 98.049 s and the time for HTTP request-response messaging was 112.723 s. By this simplification however the transmission time is depreciated in favor of the server.

Finally, both values just have to be extrapolated.

[7] Lower level executions on the server that were not captured with the used profiling technique are ignored for simplification purposes.

Chapter 6

Summary, Conclusion and Outlook

This thesis discussed the possibilities of utilizing the computational power of web clients and presented approaches to the design of simple interfaces for managing and controlling large clusters of so-called workers. Several important, technological factors for a successful deployment were discussed in this context that outlined both problems and opportunities.

In addition, instruments and methods have been presented for measuring and predicting the maximal summable power behind such vague clusters. Subsequently, a testing system and a framework prototype were build, based on adapted algorithms such as MapReduce. Due to the application under real conditions, it was possible to obtain extensive test data and evaluate crucial metrics for these components.

The results, and insights, gained within these work, as well as points of concern and further ideas, are finally summed up and discussed.

6.1 Summary and Conclusion

Prior to this work there was a discussion whether distributed computations with web clients do make a sense and if so, for what field of application. The main criticisms were certainly the total achievable power and long delays resulting from data transmission. In addition, it was unclear in which way a browser would react to unusually high load.

For this reason it was suggested that website visitors could leave the site and flee from browser crashes or not responding surfaces. In addition to this high potential risk, other difficulties were seen in the communication and the failure rate of the workers.

All of the mentioned problems occurred in the course of this work. The following section is about how they got solved or why they could not be solved.

6.1.1 Summary

Performance The all in all achievable performance of the JSMapReduce framework can best be illustrated by referring again to the comparison in table 5.7. In order to compare only the mere processing power of the Java and the JavaScript processes, all additional delays that occurred apart the computation, including the data transmission, are initially ignored.

A surprising fact, which results in this way from table 5.7 is that the mere processing in JavaScript was, on average, lightly faster than the computation with Java:

- A Java program needed 26.864 ms for the 'visits per month' calculation of an one GB sized server log file
- One JSMapReduce worker needed 26.013 s for the 'visits per month' calculation of an one GB sized server log file

Even if the result is affected by a lack of optimization of the Java code, it is, nonetheless, an impressive demonstration of the potential power that is possible to achieve with JavaScript workers. This result is definitely a resounding success, since worker processes reached this equal result, while running parallel to a game and in the browser.

The downside of this result is, however, that it can not be achieved without high costs for overhead.

Overhead Preceding sections demonstrated that a low overhead is a crucial factor to success. The differences between the estimated and the actual reached results, however, show that the overhead is significantly higher than expected. Referring to table 5.7, only the runtime of the scheduler was, for example, 20 times as high as the estimated value.

The values for the preprocessing (see table5.7) were also significantly greater than expected. The values demonstrate that even for less complex and singular tasks the duration of the preprocessing can be already higher than the total computation.

Data Transmission In order to estimate the expected overhead, caused by data transfer, section 5.1.1 on page 40 dealt with basic insights into network delays and a superficial overview of the German broadband network.

Finally, it can be said that the transmission times were below the estimated values. Most of the clients had fast Internet access which allowed the transport within a reasonable time. Some very fast clients fetched twice or three times as many jobs as the average client and thus improved the good, overall impression.

The transport of text files has been particularly favored, due to the gzip compression. The values for the data transmission demonstrated that, in the case of text files, the data transfer does not necessarily have to be the decisive factor for an efficient deployment.

At the same time, however, the server caused some strong delays, that thwarted a smooth data flow. The server-side profiling revealed that the determination and retrieval of data or in other word the 'getChunk' method was responsible for about 80% of the total processing time in the scheduler(see page 66). These time-consuming overhead cost additional 98 seconds and thereby doubled the actual value for fetching data to 210 s compared to the estimation of 113 s (see table 5.7).

Browser A critical consideration in the system's design concerned the size of the chunks, that is, the amount of data that is large enough to compensate transmission and processing overhead, while being still manageable by a browser. This value was determined, more or less by trial and error, to about 1.5 MB, which had rarely noticeable stability impact on the involved browsers. In order to support dynamically determined chunk sizes, the lines of the log file were stored separately from each other.

The second uncertainty were the possibilities in the use of the web worker technology. In the test, however, already 98% of the visitor's browsers supported this technology. In addition, the tests showed that web workers are already stable and easy to set up.

The browser became a real test winner with the user's evaluation. The survey results showed that there were occasional glitches but not a single crash. Most of the participants endorsed the deployment of workers for certain application fields (see page 67).

Communication The bidirectional communication could be solved with 'Comet' in the form of XHR Long Polling (see page 24). Only the long waiting times that used to emerge from HTPP requests are an incentive to use a more data stream oriented solution.

Worker Failures Client errors are a persistent problem for the framework. They appeared only occasionally in the tests but were difficult to determine during runtime. The web workers triggered a total of 13 times the onerror event of their API (see page 53). The failures, however, could not be associated with a specific browser or OS. All faulty workers, however, had in common that at least one of the two mentioned components was outdated.

In addition to these really buggy workers, some problems could be recognized for interrupted and slow workers. Methods for the detection and treatment of such problems were discussed in the 'Correlation' section on the page 57 et seq.

Overall, the re-execution of jobs proved to be a good counteraction.

6.1.2 Conclusion

At the beginning of this thesis, many implementation related decisions were dependent on unknown or vague values and characteristics. The following subsection is an attempt to formalize the obtained results and interactions and thereby provide insights to some of the crucial factors for both a successful deployment and a further development of the framework.

Efficient Deployment While it is expected from a single-threaded Java program to work only with approximately constant processing power, the high parallelization of tasks in the JSMapReduce framework, theoretically, allows a inversely proportional decrease in runtime for an increasing worker number. The decrease is however limited to the worst-case computation time of a single worker. More specifically, if one job has 667 data chunks and each chunk is processed absolutely parallel by a different worker, the ideal, total processing time is equal to the average[1] processing time of one chunk.

Since the computational power of the server, however, does not scale with its connected clients, the overall performance is damped. This effect is described in the server profiling section on page 64. The section introduced the term 'damping' to describe the impact of multiple concurrent connected clients on the server's performance. Damping thereby defines the ratio between a number of expected and actual allocated jobs, for a given number of workers.

A second limiting factor is the transmission time. The data transport, however, can also be parallelized but is limited again by the server's own bandwidth and performance.

The damping ratio, that is, the relation between the concurrent worker number and decreasing transmission rate is illustrated in the chart 5.13 on page 65 and was approximated for the used virtual machine with the polynomial $y = 0,0002x3 - 0,0514x2 + 4,0053x - 4,2869$. The specified range of values here is the interval [1,64].

[1]On condition that all workers are equally fast

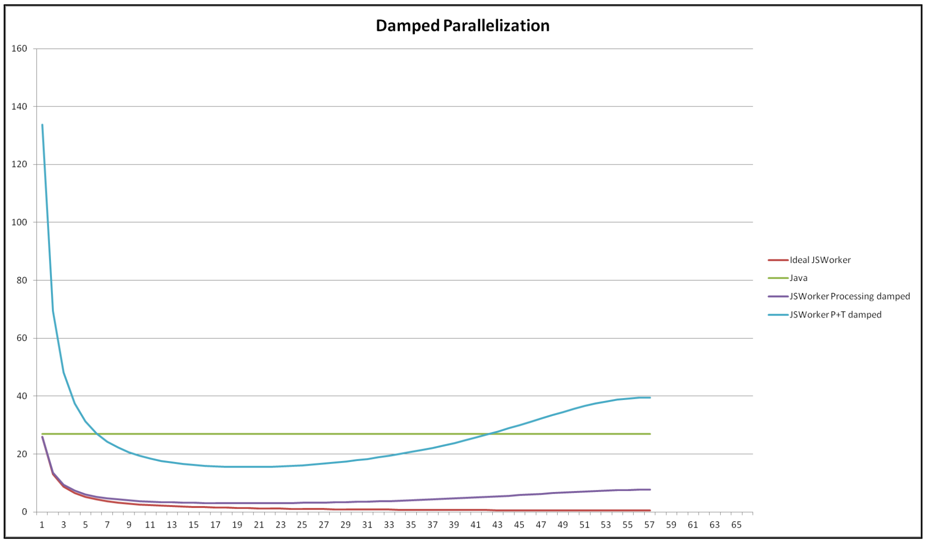


Figure 6.1: Damped Parallelization

Diagram 6.1 illustrates the total processing time in seconds (y-axis) for an increasing worker number (x-axis), with respect to the test case and the measured values. The red line thereby stands for an ideal process, that is, without performance loss for arbitrary worker numbers. Its total processing time is, furthermore, reduced with increasing worker number and approaches inversely proportional to the average processing time of a single chunk (see explanation above).
In comparison to this parallelized execution, the constant working Java program is displayed with the green line.
According to the described damping, the throughput is actually reduced for an increasing worker number. This effect is taken into account, once, for the mere processing time(purple) and once for the combination of transmission time and processing time(light blue).

It can be seen very clearly that at one point the parallelization goes into reverse and the throughput decreases. If the system is overloaded, the total calculation is even slower than that of a single worker (≈ 300 workers).

The optimal number of workers is characterized by the minimum of the function and is located at about 20 concurrent workers. At this point, the calculation with JSMapReduce is about 30% (10s) faster than a simple Java program.

The overhead that is generated in the preprocessing was, however, completely neglected in the above chart. Since the server can read data only sequentially, further restrictions are associated with the scheduling process.

By formalizing these findings and considerations, the total process time can be (simplified) described with the following formula.

T : Total Processing Time
O : Overhead
tw_i : the time it takes for a worker on average to calculate chunk i
tr_i : the time it takes to transfer the data of chunk i
C : Chunks
W : Parallelization (worker count)
D() : Damping function

$$T = O + \frac{\sum_{i=1}^{C}(tw_i + tr_i)}{W * D(W)}$$

By applying the above formula to the results of the test runs, the following representation is obtained.

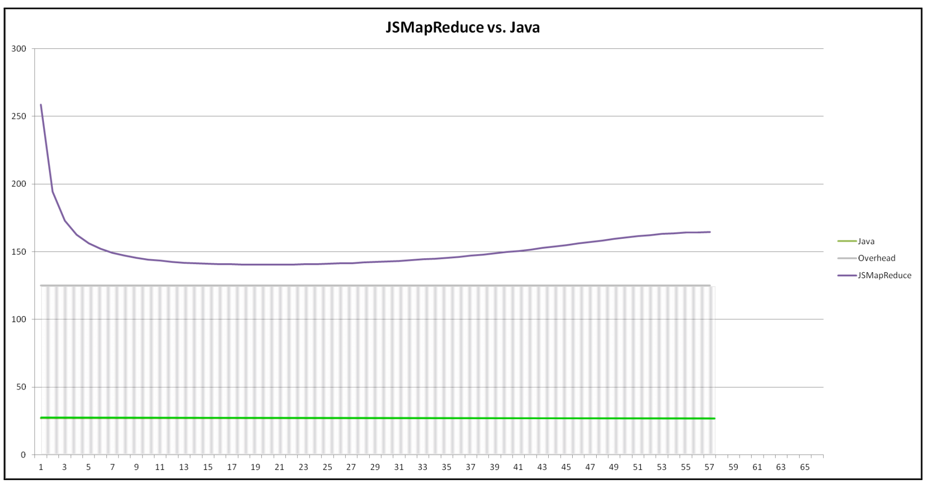


Figure 6.2: JSMapReduce vs. Java I

The diagram 6.2 shows the final comparison between JSMapReduce and a conventional approach, here, with the example of a Java program. The particular values for varying worker numbers were calculated with the above formula. The overhead, resulting from preprocessing steps such as requesting and request handling, is graphically indicated through the shaded area.

As can be seen, the overhead causes the JSMapReduce approach to be not competitive at all. Hence, the question that must be answered is whether there is a situation the JSMR framework can prevail against conventional approaches.

Due to the required overhead, the system, ultimately, was not nearly as efficient as it could be. The mere processing times as well as the stability of the system, on the contrary, have already been very well. Even the transmission time for compressed text files was acceptable.

Assuming that the web server would not have written to a text file but stored its entries directly into the database, and by using the measured reading values instead of the fetching values, the following course of curve is obtained.

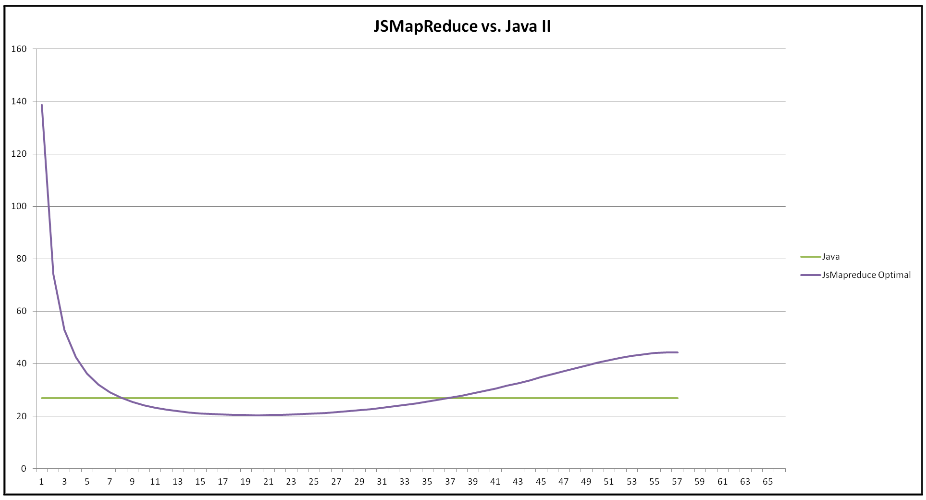


Figure 6.3: JSMapReduce vs. Java II

In this chart the JSMapReduce approach is already faster at 8 parallel working web clients. Considering that the system is just in its prototype stadium and is run on a virtual machine, the results are very surprising and encouraging. The current trends indicate that all used techniques continue to undergo a rapid development and an upper performance limit is not foreseeable.

But even now, further optimizations are possible and reasonable. The next section will finally outline some of them.

6.2 Optimization

The workers demonstrated impressively what is possible with today's browsers. The server application, however, had some unacceptable performance issues and created a relatively high overhead which in turn had a negative impact on the overall performance. With the above formula some further parameters can be determined and serve as a starting point for the optimization steps.

The following section is a brief proposal for alternative software components and possible modifications for a future development. In particular, two possibilities for optimization are proposed, that came up in the course of this thesis and are of great importance. The first optimization addresses the problem of having little to no worker available, while the second optimization deals with the problem of having too much workers and choosing those with the highest quality in a 'long term'.

6.2.1 General

LAMP The used LAMP system was primarily chosen for its good reputation and the suitability for quick prototyping. LAMP does in addition provide low level entry barriers for the development, due to its high platform independence. Some decisions in the framework's development were made in favor of this advantages and against high performance. At the same time, many open source alternatives were neglected that would have been suitable replacements. A great number of theses alternatives is designed for high performance tasks and achieves good benchmark results in their respective field of application.

In addition, the virtual test server, was barely applicable or optimized for many visitors. A better web server, combined with an extra database server would certainly have improved the results. In particular, the damping would have been a different which in turn would have led to a higher efficiency in parallelization, thus a better total performance.

In some cases, a well-configured Apache could have led already to a considerable increase in performance. The threading for example could be tested and compared with different modules.
The nginx server should be mentioned at this point as a promising alternative to the Apache. Its popularity is steadily growing and it has a very good reputation especially in the high performance sector.

Reading from the MySQL database was the major bottleneck of the system. This was probably due to the data granularity, that is, the chunk size and the database schema. The number of competing processes and the hard disk speed are additional factors.
It is possible, that a traditional relational DBMS (RDBMS) was not the best solution for the task. An open source solution that definitely has to be mentioned at this point is HBase. HBase is a DBMS that pursues a basically different strategy, referred to as 'NoSQL'. HBase was designed after Google's BigTable and gained great attention through the use in Hadoop. Meanwhile, large companies like Facebook rely on it and accelerate and support its development process. The same application field, together with the fact that it is 'big data'-approved, make it an excellent candidate. At the same time, however, must be admitted that a major advantage of HBase is the management of distributed data, while the JSMapReduce framework was designed completely centralized for a good

reason.

The last LAMP component that must be considered is PHP. PHP enables a flexible programming and provides various optimization possibilities. Nonetheless, during the development process, PHP revealed some serious weaknesses. PHP complicated things that else have been no problems. To overcome these problems, such as the reading of large text files, the latest framework version already makes use of Java hooks dealing with time sensitive tasks.
However, PHP has established itself well, in the Web. The developer community and companies like Facebook drive the development forward and work continuously to improve its performance. A promising product of such effort is HipHop. With this compiler, it is possible to convert PHP into C++ code. But even this is not a free ticket for careless programming or unrestricted use of PHP. Rasmus Lerdorf, the inventor of PHP, for example, is quoted as saying on optimization in general and with HipHop:
'Even if you double the execution speed of something that is 10% of your overall request cost, that is only a 5% overall improvement. If on every request you are hitting memcache/postgresql/mysql 10 times and spending a lot of time in system calls, don't expect miracles from HipHop.'[18]

Framework But not only the LAMP stack can be optimized. The framework itself provides lots of opportunities for improvements and changes. In contrast to very specific things, such as an enhanced security and protection system, encoded chunks or a more reliable authentication, optimizations should start at the very core of the framework. But the framework could also be extended with an external cache-server such as memcached. Memcached could improve the system by caching the latest distributed chunks in order to enable a faster redundant allocation.

Moreover, the preprocessing and the chunk size should be reviewed to determine the ideal tradeoff between flexibility and efficiency.

The server push concept could also be redesigned, even if the used technique was very stable and performed well. The WebSocket API provides for bi-directional, full-duplex communication and is standardized, thus supported by most modern browsers.

As an alternative to the conversion with , the whole framework could of course be completely rewritten in a different programming language. JavaScript on the server in combination with Node.js is just one new concept that is gaining more and more popularity. This homogeneity would, for example, offer benefits for maintenance and further development.

Worker Code The worker performance turned out to be very good. Further tests could measure how the worker thread performs in simple pages or in the background of massive applications, such as complex games. Such tests would maybe reveal weaknesses and optimization opportunities.

Other changes would include the generation of worker code in the server script. As a result, certain parts such as the map and reduce functions could be set with the beginning of a session and would not have to be sent with each chunk anymore. This would, however, embody a violation of the basic flexibility concept of the framework.

6.2.2 Hybrid

One crucial question that accompanied the project was how to compute a fully preprocessed job without having the needed visitors/clients. To avoid an idle system in such a case, there is a simple yet effective solution. The technique combines the advantages of the framework with those of a stand alone application by turning the server itself into a worker. Since it would not make sense for the most servers to use a browser and open a tab for this purpose, the hybrid is designed different. The current prototype was made in Java. One HTTP based and really straightforward draft is outlined in listing 6.1.

```
1  public class Worker implements Observer{
2
3    Bridge bridge;
4    String id;
5
6    [...]
7
8    public Worker(){
9       this.bridge = new Bridge(this);
10   }
11
12   public void getJobHTTP(){
13     if(bridge.connect("localhost", 80)){
14         String s = "GET "+"/mapreduce/scheduler.php?wnr="+this.id +
               " HTTP/1.0"+"\r\n\r\n";
15         bridge.sendMessage(s);
16     }
17   }
18
19   public void update(Observable arg0, Object arg1) {
20       String message = (String) arg1;
21       this.doJob(message);
22   }
23   [...]
24 }
```

Listing 6.1: Hybrid: Java Code

The hybrid worker, described in listing 6.1, uses the socket interface of the bridge (see page 34) to establish a connection with the Apache on port 80. Through this connection, simple HTPP requests like the GET request in line 14 are sent.

At the same time, the worker registers as an observer of the bridge so that its update method is called whenever new data arrives. The data is then passed to the doJob method and computations are started.

The hybrid approach could be improved by providing the worker direct access to the database, this ,however, could create conflicts and would break the separation of master and client. The simple server worker, in contrast, can simultaneously operate with conventional workers, without risking the determinism or the regular execution. More than that, the system becomes even faster whenever other Java workers are added to the cluster. Such workers can be started from, e.g., workstation PCs in the same network but also volunteers that are connected over the Internet.

Through efficient implementation of the hybrid worker, one major flaw of the system will be eliminated, while the performance is massively increased.

6.2.3 Allocation Strategy

Maintaining an unnecessarily large cluster of workers is a big misconception, according to the resulting performance damping (see page 64). For this reason, the ability to distinguish between high quality and useless workers cannot be overestimated. The section on intra-procedural variations within workers has described why this is not a trivial task. Based on the insights gained within the tests and this section, the following listing provides an optimized allocation strategy.

1. exclude all workers with outdated browser or OS (p. 53)
2. exclude all mobile devices (p. 31)
3. assign to each remaining worker one single chunk
    (a) store the process time of the fastest worker as a reference value
4. workers that trigger the onerror event sign themselves out and are not allowed to register again for the job (p. 51)
5. send the reference values for the transmission and processing time with with every following chunk
    (a) worker have an internal counter and a tolerance factor
    (b) if the reference value is exceeded above the tolerance, they increment their own counter (p. 61)
    (c) if their counter reaches the limit of $(chunkcount/workercount) * 0.2$, they will sign out for the current job or start a timeout (p. 61)
6. the system resets all uncompleted chunks
7. the system sets slow processed chunks to unassigned (see listing 4.2.4 on page 32)

The presented algorithm follows an approach of early recognition and tries to relieve the system by giving workers the possibility to unregister themselves. It is optimized for large-sized clusters, thus may has a negative impact on small clusters. Since the allocation of jobs is an essential factor for the system's overall stability and performance, this basic approach should be considered and developed further in later implementations.

6.3 Outlook

The factor computing power will grow in importance in the future - not only for large-scale enterprises but also smaller institutions or single persons. This situation is favored by the progressive development in software engineering and growing sizes of modern applications. As long as the requirements of computational tasks surpass the available processing power and requirements grow faster than the performance of today's CPUs, the economical use of available resources will be an essential task.

Even these days computational power is an economic factor. Bitcoins, an alternative, digital currency demonstrate this fact in an impressive way since their value creation process, called bitcoin mining, relies on selling computing cycles.

The introduced framework provides an innovative approach, for taking advantage of potential resources while balancing the computational load between server and client. Whether such a solution can prevail in the long term is dependent on many factors and can nearly impossible predicted at the present time.

Nevertheless, today's developments indicate, that the performance of the used components will continue to increase and bottlenecks like slow hard drives or low bandwidth will soon be history. Even smart phones go through a development process that would have made every computer in the last 10 years envious. Soon, quad-core processors and data flat rates with high bandwidth will be common practice in this sector.

At the end, the user will play a major role, since he decides whether and in which way such a system is used and permitted. The confidence is strengthened however by the fact that projects such as SETI@Home or BOINC already proved that the voluntary offering of own resources can lead to a similar culture as open source software.

Bibliography

[1] BITKOM: *Mehr Bandbreite im Breitband.* 19.07.2011. – URL http://www.bitkom.org/de/markt_statistik/64042_68649.aspx

[2] COMPUTER BILD Digital GmbH: *Der große DSL-Geschwindigkeitstest - DSL Speed testen - COMPUTER BILD.* 2012. – URL http://dsl-speedtest.computerbild.de/

[3] Crockford, Douglas: *The application/json Media Type for JavaScript Object Notation (JSON).* Internet RFC 4627. jul 2006. – URL https://github.com/douglascrockford/JSON-js

[4] Dean, Jeffrey ; Ghemawat, Sanjay: MapReduce: Simplified Data Processing on Large Clusters. In: *OSDI 2004*, URL http://www.usenix.org/events/osdi04/tech/dean.html, 2004, S. 137–150

[5] Deveria, Alexis: *When can I use Web Workers?* 2011. – URL http://caniuse.com/webworkers

[6] DSLWEB, Ehninger AG: *DSL Marktübersicht - die DSL Kundenzahlen im 4. Quartal 2011.* 01.10.2011. – URL http://www.dslweb.de/dsl-marktuebersicht.php

[7] DSLWEB, Ehninger AG: *DSL 6000 Speed Test: Soviel Speed bringt DSL 6000 in der Praxis.* 2012. – URL http://www.dslweb.de/dsl-6000-speed-test.php

[8] DSLWEB, Ehninger AG: *DSL Anschluss Test: So schnell ist ein DSL Anschluss in der Praxis.* 2012. – URL http://www.dslweb.de/dsl-anschluss-speed-test.php

[9] François Bry, Christoph Wieser Tim F. ; Zauner, Harald: *Webinformationssysteme,2011.* – URL http://www.pms.ifi.lmu.de/lehre/webinfosys/11ws12/. – Lecture in the winter semester of 2011/2012 at Ludwig-Maximilians-Universität München

[10] Garrett, Jesse J. ; LLC, Adaptive P. (Hrsg.): *Ajax: A New Approach to Web Applications.* 18.02.2005. – URL http://www.adaptivepath.com/ideas/ajax-new-approach-web-applications. – Zugriffsdatum: 20011

[11] Gemius, MENA: *Web browsers groups details - gemiusRanking MENA*. – URL http://www.rankingmena.com/en/rankings/web-browsers-details.html

[12] Google Inc.: *What is Google Chrome OS?* 18.11.2009. – URL http://www.chromium.org/chromium-os

[13] Google Inc., Webmasters Google D.: *Web Authoring Statistics: Scripting*. 2005. – URL http://code.google.com/intl/de-DE/webstats/2005-12/scripting.html

[14] Grigorik, Ilya: *Collaborative Map-Reduce in the Browser*. 03.03.2009. – URL http://www.igvita.com/2009/03/03/collaborative-map-reduce-in-the-browser/

[15] He, Bingsheng ; Fang, Wenbin ; Luo, Qiong ; Govindaraju, Naga K. ; Wang, Tuyong: Mars: a MapReduce framework on graphics processors. In: *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*. New York, NY, USA : ACM, 2008 (PACT '08), S. 260–269. – URL http://doi.acm.org/10.1145/1454115.1454152. – ISBN 978-1-60558-282-5

[16] Ilya Kantor: *Events and timing in-depth*. 2011. – URL http://JavaScript.info/tutorial/events-and-timing-depth

[17] John Resig: *Computing with JavaScript Web Workers*. 21.06.2009. – URL http://ejohn.org/blog/web-workers/

[18] Jolie O'Dell, ReadWriteWeb: *PHP Creator Asks, Is Facebook's HipHop Just a 'Nifty Trick'?* 2012. – URL http://www.readwriteweb.com/archives/rasmus-lerdorf-php-hiphop-facebook.php

[19] Ken Kovash, Mozilla F.: *What is Firefox's Market Share?* 19.03.2009. – URL http://blog.mozilla.com/metrics/2009/03/19/what-is-firefoxs-market-share/

[20] Kovas Boguta, AOL Inc.: *Hadoop & Startups: Where Open Source Meets Business Data*. 17.07.2011. – URL http://techcrunch.com/2011/07/17/hadoop-startups-where-open-source-meets-business-data/

[21] LTD, NETCRAFT: *October 2011 Web Server Survey*. 06.10.2011. – URL http://news.netcraft.com/archives/2011/10/06/october-2011-web-server-survey.html. – In the October 2011 survey we received responses from 504,082,040 sites, a growth of nearly 3.8main web

[22] Maone, Giorgio: *NoScript : Add-on for Firefox*. 2012. – URL https://addons.mozilla.org/en-US/firefox/addon/noscript/?src=cb-dl-users. – It allows JavaScript, Java and other executable content to run only from trusted domains of your choice

[23] Maurice de Kunder: *WorldWideWebSize.com — The size of the World Wide Web.* 2012. – URL http://www.worldwidewebsize.com/. – The size of the World Wide Web is estimated here every day. The Indexed Web contains at least 8.6 billion pages

[24] Mike Belshe, Google I.: *A Client-Side Argument for Changing TCP Slow Start.* 11.01.2010. – URL https://6541078575799853287-a-chromium-org-s-sites.googlegroups.com/a/chromium.org/dev/spdy/An_Argument_For_Changing_TCP_Slow_Start.pdf

[25] Miniwatts Marketing Group: *World Internet Usage Statistics News and World Population.* 2012. – URL http://www.internetworldstats.com/stats.htm. – Internet World Stats, Population and Internet Users in all countries and regions of the world. The Internet Big Picture.

[26] NEWS, BBC: *Most websites' failing disabled.* 05.12.2006. – URL http://news.bbc.co.uk/2/hi/technology/6210068.stm

[27] Nicholas C. Zakas: *What determines that a script is long-running?* 05.01.2009. – URL http://www.nczonline.net/blog/2009/01/05/what-determines-that-a-script-is-long-running/

[28] Nicholas C. Zakas: *What is a non-blocking script?* 10.08.2010. – URL http://www.nczonline.net/blog/2010/08/10/what-is-a-non-blocking-script/

[29] Olav Junker Kjær, Opera Software ASA: *Timing and Synchronization in JavaScript.* 27.02.2007. – URL http://dev.opera.com/articles/view/timing-and-synchronization-in-JavaScript/

[30] Oracle Corporation: *MySQL 5.0 Reference Manual : 11 Data Types.* 2012. – URL http://dev.mysql.com/doc/refman/5.0/en/data-types.html

[31] Oracle Corporation: *MySQL 5.1 Reference Manual : What is MySQL?* 2012. – URL http://dev.mysql.com/doc/refman/5.1/en/what-is-mysql.html

[32] Ranger, Colby ; Raghuraman, Ramanan ; Penmetsa, Arun ; Bradski, Gary ; Kozyrakis, Christos: Evaluating MapReduce for Multi-core and Multiprocessor Systems. In: *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture.* Washington, DC, USA : IEEE Computer Society, 2007 (HPCA '07), S. 13–24. – URL http://dx.doi.org/10.1109/HPCA.2007.346181. – ISBN 1-4244-0804-0

[33] Rethans, Derick: *Profiling PHP Applications.* 2011. – URL http://derickrethans.nl/talks/profiling-zendcon11.pdf

[34] Rich Miller, Data Center Knowledge: *Google Data Center FAQ, Part 2*. 26.08.2008. – URL http://www.datacenterknowledge.com/google-data-center-faq-part-2/

[35] Ryza, S. ; Wall, T.: MRJS: A JavaScript MapReduce Framework for Web Browsers, URL http://www.cs.brown.edu/courses/csci2950-u/f11/papers/mrjs.pdf, 2010

[36] StatCounter: *StatCounter Global Stats - Browser, OS, Search Engine*. – URL http://gs.statcounter.com/. – Tracks the Market Share of Search Engines, Browsers and Operating Systems including Mobile from over 15 billion monthly page views.

[37] The Apache Software Foundation: *About Page*. 2012. – URL http://couchdb.apache.org/

[38] The Apache Software Foundation: *About Page*. 2012. – URL http://hadoop.apache.org/

[39] The Apache Software Foundation: *Apache Performance Tuning - Apache HTTP Server*. 2012. – URL http://httpd.apache.org/docs/2.0/misc/perf-tuning.html

[40] The Apache Software Foundation: *Documentation: Apache HTTP Server - The Apache HTTP Server Project*. 2012. – URL http://httpd.apache.org/docs/

[41] The Apache Software Foundation: *Multi-Processing-Module (MPMs)*. 2012. – URL http://httpd.apache.org/docs/2.2/mpm.html

[42] The Nielsen Company: *February 2011: Top U.S. Web Brands and Travel Sites*. 16.03.2011. – URL http://blog.nielsen.com/nielsenwire/online_mobile/february-2011-top-u-s-web-brands-and-travel-sites/. – Nielsen, Consumer Insights, Global Media and Consumer Research

[43] The Nielsen Company: *What Americans Do Online: Social Media And Games Dominate Activity*. 2010. – URL http://blog.nielsen.com/nielsenwire/online_mobile/what-americans-do-online-social-media-and-games-dominate-activity/

[44] The PHP Group: *About PHP*. 2012. – URL http://www.php.net/

[45] TheCounter.com (global statistics): *Usage share of web browsers*. – URL http://en.wikipedia.org/wiki/Usage_share_of_web_browsers#TheCounter.com_.282000_to_2009.29

[46] W3C: *Web Workers*. 2012. – URL http://www.w3.org/TR/workers/

[47] W3Techs: *Usage of client-side programming languages for websites*. 07.06.2012. – URL http://w3techs.com/technologies/overview/client_side_language/all. – W3Techs shows statistics and trends in the usage of client-side programming languages.

[48] W3Techs: *Usage of server-side programming languages for websites*. 2012. – URL http://w3techs.com/technologies/overview/programming_language/all. – W3Techs shows statistics and trends in the usage of server-side programming languages

[49] WHATWG: *Is this HTML5?* 2012. – URL http://www.whatwg.org/specs/web-apps/current-work/multipage/introduction.html#is-this-html5?

[50] WHATWG: *Web workers HTML Standard*. 2012. – URL http://www.whatwg.org/specs/web-apps/current-work/multipage/workers.html

[51] Whitson Gordon, Gawker Media: *Browser Speed Tests: Firefox 7, Chrome 14, Internet Explorer 9, and Opera 11.51*. 27.09.2011. – URL http://lifehacker.com/5844150/browser-speed-tests-firefox-7-chrome-14-internet-explorer-9-and-more

[52] Wikipedia: *Shannon-Hartley theorem - Wikipedia, The Free Encyclopedia*. 2012. – URL http://en.wikipedia.org/w/index.php?title=Shannon%E2%80%93Hartley_theorem&oldid=495771180. – [Online; accessed 26-June-2012]

Index

W

X

List of Tables

List of Figures

Listings

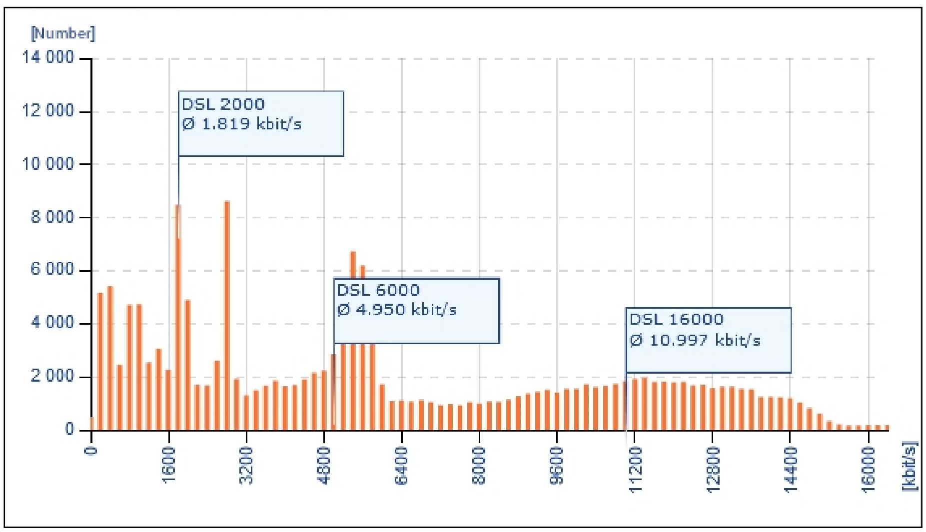


Figure 4: Bandwidth Statistic