TAMPERE UNIVERSITY OF TECHNOLOGY

PETRI HAUTAMÄKI

IMPROVING WEB APPLICATION SCALABILITY WITH ADVANCED CACHING TECHNIQUES

Master's Thesis

Examiner: Adjunct Professor Ossi Nykänen
Examiner and topic approved
by the Council of the Faculty of Computing and Electrical Engineering
on 5th November 2014

TIIVISTELMÄ

TAMPEREEN TEKNILLINEN YLIOPISTO
Tietotekniikan koulutusohjelma
HAUTAMÄKI, PETRI: Internet-sovellusten skaalautuvuuden parantaminen kehittyneitä välimuistitustekniikkoja hyödyntämällä
Diplomityö, 56 sivua, 9 liitesivua
Marraskuu 2014
Pääaine: Hypermedia
Tarkastajat: Dosentti Ossi Nykänen
Avainsanat: taustajärjestelmä, Internet-sovellus, skaalautuvuus, vertikaalinen skaalautuminen, horisontaalinen skaalautuminen, välimuisti, Edge Side Includes, Grails

Tämän diplomityön pääasiallinena tarkoituksena oli tutkia ja valita parhaat tavat Internet-sovellusten skaalautuvuuden parantamiseksi erityisesti korkeiden liikennepiikkien aikana. Sovellusten skaalautuvuuden konkreettisimmaksi mittariksi valittiin sovelluksen maksimaalinen läpisyöttökyky.

Yleisten Internet-sovellusten tehokkuussuositusten perusteella lupaavimpana ratkaisuna läpisyöttökyvyn parantamiseen nähtiin välimuistin käytön tehostaminen. Tämän pohjalta todelliseen sovellukseen implementoitiin kaksi kehittynyttä välimuistitustekniikkaa: sisältöosasten välimuistittaminen Edge Side Includes (ESI) -merkkauskieltä hyödyntämällä sekä käyttäjäryhmäkohtaisten HTTP-vastausten välimuistittaminen Servlet-suodattimen avulla.

Edge Side Includes -tuen implementoinnin jälkeen sovelluksen etusivun maksimaalisen läpisyöttökyvyn mitattiin olevan noin kolminkertainen alkuperäiseen verrattuna. Vaikka tämä katsottiin jo hyväksi parannukseksi, vaikutti etusivun rungon renderöiminen olevan tässä vaiheessa niin raskasta, että se turhaan rajoitti ESI-ratkaisun todellista potentiaalia.

Rungon renderöimistehokkuuden parantamiseksi sovelluksen ympärille luotiin Servlet-suodatin, jonka avulla etusivun eri versioita välimuistitettiin parin sekunnin ajan. Ratkaisun ansiosta sovelluksen prosessointitarve väheni merkittävästi erityisesti korkeiden liikennepiikkien aikana, jonka seurauksena etusivun läpisyöttökyvyksi saatiin nyt noin kymmenkertainen arvo alkuperäiseen verrattuna.

Edellisten lisäksi sovelluksen maksimaalinen kokonaisläpisyöttökyky mitattiin ajamalla realistista korkean liikenteen piikkiä kuvaava simulaatio sekä optimoimattomalle että optimoidulle sovellukselle. Mittausten perusteella optimoitu sovellus suoriutui simulaatiosta noin 2.3 kertaa alkuperäistä paremmin.

Diplomityöprojekti voidaan katsoa kokonaisuudessaan onnistuneeksi, sillä sen ansiosta alkuperäisen sovelluksen skaalautuvuutta saatiin parannettua merkittävästi. Lisäksi projektin aikana löydettiin monia jatkokehitysajatuksia skaalautuvuuden parantamiseksi entisestään tulevaisuudessa.

ABSTRACT

TAMPERE UNIVERSITY OF TECHNOLOGY
Master's Degree Programme in Information Technology
HAUTAMÄKI, PETRI: Improving web application scalability with advanced caching techniques
Master of Science Thesis, 56 pages, 9 Appendix pages
November 2014
Major: Hypermedia
Examiner: Adjunct Professor Ossi Nykänen
Keywords: back end, web application, scalability, vertical scaling, horizontal scaling, cache, Edge Side Includes, Grails

The main purpose of this thesis was to research and select the most usable ways of improving web application scalability especially during high traffic peaks. The most concrete meter for the application scalability was defined to be the maximum throughput of the application.

From the general web application performance guidelines, caching was seen as the most promising solution for improving the maximum throughput of the application. Because of this, a couple of advanced caching techniques were implemented into a real life application: caching of content fragments with Edge Side Includes (ESI) markup language and caching of user group specific HTTP responses with a Servlet filter.

After implementing the Edge Side Includes support into the application, the maximum throughput of the application front page was measured to be roughly 3 times the original. Though this was considered to be a good improvement, it seemed clear that the rendering process of the front page response skeleton was unnecessarily heavy at this point, therefore limiting the true potential of the ESI solution.

In order to improve the performance of the response skeleton rendering process, a Servlet filter was configured to cache the different response skeletons to a local cache for two seconds, thus reducing the need for processing in the application especially during high traffic peaks. With this improvement, the maximum throughput of the application front page now reached approximately tenfold value compared to the original version.

In addition, the maximum total throughput of the application was measured under a realistic simulation of a high traffic peak before and after the optimization steps. As a result, the optimized application seemed to perform approximately 2.3 times better than the original.

In the end, the thesis project can be considered successful due to the noticeable improvement in the application scalability. Furthermore, various additional improvement possibilities were discovered during the project, which could help to improve the application scalability even more in the future.

PREFACE

This thesis was written for the company Conmio Oy in order to find usable ways of improving the scalability of the company's web applications. The thesis project was started originally in the beginning of June 2013, yet the final topic was found later during January 2014, when a concrete issue in a real life web application scalability was encountered. The thesis examiner, Adjunct Professor Ossi Nykänen, and the topic were approved by the Council of the Faculty of Computing and Electrical Engineering on 5th November 2014.

I want to thank especially the staff of Conmio Oy for providing me a suitable working gear, environment and guidance during the thesis project. Special thanks to Ville Viskari, Tatu Dufva, Antero Fagerstedt and Tommi Liittokivi for the great input concerning the research around the thesis. A very special thanks to the CEO of Conmio, Tero Hämäläinen, for allowing the thesis project to happen in the first place.

Additionally, I want to thank my official instructor and examiner Ossi Nykänen from the Tampere University of Technology for giving me a lot of valuable feedback, especially concerning the structure of the thesis. Without his feedback, the final product would not be as coherent as it is now.

Now when the thesis is finally finished, the feeling is relaxed. A long career at school is about to come to an end while another one at work is just about to begin. Along with the new career begins also the actual learning process of the new Master of Science. Especially in the area of Information Technology, where everything advances in lightning fast speeds, the true professionals can never truly rest if they want to stay as the masters of their game.

What comes to the results of the thesis, there is some anticipation in the air. Despite the most realistic load tests and simulations available, the final load test will always happen in the actual production environment by the actual end users. At the time of writing, a few new second screen applications are already out and optimized with the techniques explained in this thesis. However, the anticipated high traffic peak is still yet to come.

Tampere, 10th November 2014

Petri Hautamäki

CONTENTS

1. Introduction 1
1.1 The purpose of the thesis 1
1.1.1 Research problem 1
1.1.2 Research questions 2
1.1.3 Research methods 2
1.1.4 Scope 2
1.2 Structure 3
1.3 Stakeholders 4
1.4 Special markings 4
2. Starting point 5
2.1 Internet 5
2.1.1 Beginning 5
2.1.2 World Wide Web 5
2.1.3 Traditional Web 6
2.1.4 Mobile Web 6
2.1.5 Responsive Web 7
2.1.6 Second screen applications 8
2.1.7 Scalability 8
2.2 The basic structure of the Internet 10
2.2.1 From web pages to web applications 10
2.2.2 Client-server model 11
2.2.3 Server-side architecture 12
2.2.4 Server-side programming 12
2.2.5 Client-side architecture 14
2.2.6 Client-side programming 15
2.2.7 Multi-tier architecture 16
2.2.8 Conmio web applications 17
2.2.9 Conmio web application architecture 17
2.3 Programming frameworks 18
2.3.1 Purpose 19
2.3.2 Advantages 19
2.3.3 Disadvantages 19
2.4 Grails 20
2.4.1 Main features 20
2.4.2 Plugins 21
2.4.3 Advantages 21
2.4.4 Disadvantages 22

2.4.5 Alternative frameworks . . . 22
2.5 Web application performance . . . 22
2.5.1 Definition . . . 23
2.5.2 Performance best practices . . . 23
2.5.3 Load balancing . . . 25
2.5.4 Conmio web application performance . . . 25
2.5.5 Measuring the performance . . . 25
3. Improving the scalability . . . 27
3.1 Caching . . . 27
3.1.1 Theory . . . 27
3.1.2 Advantages . . . 27
3.1.3 Disadvantages . . . 28
3.1.4 Current utilization . . . 29
3.2 Advanced caching techniques . . . 29
3.2.1 Improvement options . . . 29
3.2.2 Prioritization . . . 30
3.2.3 Future possibilities . . . 30
3.3 Varnish . . . 31
3.3.1 Functionality . . . 31
3.3.2 Advantages . . . 31
3.3.3 Disadvantages . . . 32
3.4 Edge Side Includes . . . 33
3.4.1 Functionality . . . 33
3.4.2 Advantages . . . 34
3.4.3 Disadvantages . . . 34
3.4.4 Edge Side Includes and Varnish . . . 35
3.4.5 Edge Side Includes and Conmio Modules . . . 35
3.5 User group specific HTTP response caching . . . 35
3.5.1 Functionality . . . 36
3.5.2 Advantages . . . 36
3.5.3 Disadvantages . . . 37
4. Edge Side Includes utilization . . . 38
4.1 Optimization steps . . . 38
4.2 Performance before optimization . . . 38
4.3 Edge Side Includes optimization . . . 39
4.4 Performance after Edge Side Includes optimization . . . 40
4.5 Summary . . . 42
5. Response caching utilization . . . 43
5.1 Background . . . 43

5.2 Optimization steps . . . 43
5.3 Performance after response cache optimization . . . 44
5.4 Performance in a realistic high traffic simulation . . . 47
5.5 Summary . . . 49
6. Conclusions . . . 51
6.1 Selected optimization techniques . . . 51
6.1.1 Theoretical findings . . . 51
6.1.2 The results of the empirical studies . . . 52
6.1.3 Measurement reliability . . . 53
6.1.4 Further improvement possibilities . . . 54
6.2 The thesis project . . . 55
6.2.1 Successful parts . . . 55
6.2.2 Improvement areas . . . 55
References . . . 57
A. Edge Side Includes . . . 62
B. Wireframes . . . 63
C. Response Caching Filter . . . 64
D. Benchmark results . . . 65

TERMS AND DEFINITIONS

AB	ApacheBench. A single-threaded command line computer program for measuring the performance of HTTP web servers.
AJAX	Asynchronous JavaScript And XML. A collection of techniques for making web applications more interactive by allowing the exchange of data fragments through XML or JSON formats.
Apache HTTP server	A popular web server application.
Apache Tomcat	An open source software implementation of the Java Servlet and JavaServer Pages technologies.
Application server	A program that handles all application related operations between users and an organization's back end applications or databases.
Asynchronous	In computer programming, asynchronous events are those occurring independently of the main program flow. Asynchronous actions are executed in a non-blocking fashion, allowing the main program flow to continue processing.
Back end	The term back end refers to the operations that are performed on the server layer of the web application.
Bottleneck	A phenomenon where the performance or capacity of an entire system is limited by a single or limited number of components or resources.
Business logic	The part of the program that encodes the real world business rules that determine how data can be used and modified.
Caching	To utilize a cache by storing data to it or retrieving data from it.
Cache	Cache is generally a component that transparently stores data so that future requests for the same data can be served faster. Web cache is a mechanism for the temporary storage (caching) of web documents, such as HTML pages and images, for reducing bandwidth usage, server load and perceived lag on client's end.

Cache entry	An item in a cache.
Cache hit	If the requested information is found from the cache, it is called a cache hit.
Cache hit ratio	The percentage of all cache hits out of all requests is called the cache hit ratio.
Cache key	An unique value that is used for searching a particular cache entry from the cache.
Cache miss	If the requested information is not found from the cache, it is called a cache miss.
CDN	Content Delivery Network. A large system of servers distributed across the Internet designed to improve content availability and service performance.
Client	A client is a piece of computer hardware or software that accesses a service made available by a server, which may or may not be located on another computer.
Client-side	The term client-side refers to the operations that are performed on the client of client-server relationship in computer networking.
Conmio Cache	A Conmio-specific Grails plugin that allows caching of various kind of data, like Groovy and Java objects or markup fragments.
Conmio Devices	A Conmio-specific Grails plugin that offers an unified way of retrieving information about the current end device with some additional features.
Conmio Modules	A Conmio-specific Grails plugin that allows the modularization of a specific markup fragment and the required logic for composing that markup in the first place.
Conmio Oy	A software company that specializes to mobile web applications. The thesis is written for this particular company.
Dedicated server	A single computer in a network reserved for serving the needs of the network.

Domain model	Illustrates a meaningful conceptual class in the problem domain. Represents a real world concept instead of software components. For example, in the news domain an Article could be one of the domain classes.
Domain object	An instance of a domain model class.
Edge device	A device that provides entry point into enterprise or service provider network. Examples include routers, routing switches and multiplexers.
Edge-level	The term edge-level refers to the operations performed on the level of edge devices. Usually this means the middleware layer of the web application.
Ehcache	An open source, standards-based cache for boosting performance, offloading database and simplifying scalability.
End device	A source or destination device in a networked system. For example, an user's mobile phone is an end device, and so is a server.
ESI	Edge Side Includes. A markup language developed especially for caching and assembling dynamic Web content on edge-level.
ESI processor	A software capable of processing the complete response out of a response skeleton and content fragments.
Expensive processing	If processing is defined to be expensive, it requires a lot of system resources, like memory and CPU-time, in order to achieve the desired result.
Framework	An abstraction in which software providing generic functionality is selectively modified by additional user-written code, thus providing application-specific code and increasing productivity.
Front end	The term front end refers to the operations that are performed on the client layer of the web application.
Grails	A web application framework built on top of Java Virtual Machine.

Groovy	An agile and dynamic programming language for the Java Virtual Machine.
Horizontal scaling	The process of adjusting a system's resources according to load by increasing or decreasing the number of processing units.
HTTP	HyperText Transfer Protocol. An application protocol for distributed, collaborative hypermedia information systems.
IP	Internet Protocol. The main communications protocol in the Internet protocol suite for relaying datagrams across network boundaries.
Java EE	Java Platform, Enterprise Edition. Oracle's enterprise Java computing platform.
JMeter	An application by Apache Software Foundation designed to load and stress test functional behavior and measure performance of a web application.
JSON	JavaScript Object Notation. A simple open standard format for transferring data between computers. Despite the name, JSON is independent from JavaScript. JSON is an alternative to XML for example in AJAX.
JVM	Java Virtual Machine. An abstract computing machine for Java platform.
Load balancer	A software that distributes workload across multiple computing resources, such as computers, network links, central processing units or disk drives.
Middleware layer	In a multi-tier network architecture, middleware layer negotiates the transactions between client and server layers.
Multi-tier	In network architecture, multi-tier architecture refers to a system that contains more than two layers of abstraction. For example, a client-server network is multi-tier if there is some kind of middleware between the client and the server.
Origin server	The server machine running the web application back end code, thus being the origin of the actual application content.

Proxy server	A server that acts as an intermediary for requests from clients requesting resources from servers.
Request	In HTTP paradigm, clients send request messages to servers when they are in need of some specific data. The request can correspond to one of several methods, of which the most common ones are GET and POST.
Response	In HTTP paradigm, servers respond to client requests by sending a response message to the requesting client. The response contains status code that describes the successfulness of the request as well as the actual response content.
Response caching	A specific technique for caching HTTP responses containing any format of markup language, including HTML, XML, JSON or ESI elements.
Response skeleton	An intermediate version of a response that does not yet contain all required information. For example, the response returned to ESI processor is a response skeleton, if it requires assembling of content fragments before the complete response can be formed and returned to the client.
Scalability	The ability of a system to adjust its resources based on the current load.
Scaling	The process of adjusting a system's resources according to load. Scaling can be either vertical or horizontal.
Second screen	The term second screen refers usually to a mobile device that is used alongside a television show in order to enhance the viewing experience by allowing interaction with the show somehow. Possible interactions include voting a particular participant of the show, answering to a poll or chatting with other watchers of the show.
Server	The term server refers either to a particular application, program or software module that is capable of performing computing upon request or to the hardware platform that runs one or more of the previous.
Server-side	The term server-side refers to the operations that are performed on the server of client-server relationship in computer networking.

Servlet	A Java programming language class used to extend the capabilities of a server.
Servlet container	The component of a web server that interacts with Java Servlets, meaning the actual application. It is responsible for managing the lifecycle of Servlets, mapping URLs to corresponding Servlets and ensuring that the URL requester has the correct access rights.
Servlet filter	A Servlet filter is an object that can intercept HTTP requests targeted at web application.
SPA	Single-page application. A web application that fits on a single page with the goal of providing a more fluid user experience by updating the view dynamically without requiring an entire page reload after the initial response.
TCP	Transmission Control Protocol. One of the core protocols of the Internet protocol suite. Provides reliable, ordered and error-checked delivery of a stream of octets between programs running on computers connected through a network.
TCP/IP	Provides end-to-end connectivity specifying how data should be packetized, addressed, transmitted, routed and received at the destination through a network.
Throughput	In the scope of this thesis, the throughput value of a web application refers to the application's ability to serve requests per time unit.
Traffic	In computer networking, traffic describes the flow of data through the network.
TTL	Time to live. A parameter for limiting the lifespan or lifetime of data in a computer, network or cache.
Two-tier	In network architecture, two-tier architecture refers to a system that contains two layers of abstraction. For example, a pure client-server network contains just client and server layers.
Varnish	An HTTP accelerator designed especially for content-heavy dynamic web applications.

VCL	Varnish Configuration Language. A small domain-specific language designed to be used for defining request handling and document caching policies in the Varnish HTTP accelerator.
Vertical scaling	The process of adjusting a system's resources according to load by increasing or decreasing the performance of a single processing unit.
Web application	The most advanced version of a Web service. Can contain dynamically tailored representation of hypertext data for each user.
Web application framework	A software framework designed to support the development of dynamic web sites, web applications, web services and web resources.
Web server	The term web server can refer to either the hardware or the software that delivers web content that can be accessed through the Internet. Its primary function is to store, process and deliver web pages to clients.
XML	Extensible Markup Language. A markup language that defines a set of rules for encoding documents in a format that is both human-readable and machine-readable.

1. INTRODUCTION

1.1 The purpose of the thesis

The main purpose of this thesis is to research and find the most usable ways of improving web application scalability especially during high traffic peaks, thus making the applications more reliable under varying traffic conditions. In practice, these improvements would remove the need for extra monitoring concerning the anticipation of the next high traffic peak, which can be difficult especially when multiple similar applications are online simultaneously. Furthermore, these improvements would remove the need for manual scaling of the resources, which is a dull, repetitive task and costs unnecessary time, money and resources for the company.

1.1.1 Research problem

All web application logic, whether executed server-side, client-side or somewhere in-between, has a certain upper limit for performance, which is dependent of the hardware and software executing it. Since the company has a long history with mobile web applications, it is assumed that both server-side and client-side logic are already relatively well optimized by utilizing basic caching mechanisms, suitable response headers, image sprites and well-structured code overall. Therefore, the possible performance improvement achieved through this kind of basic optimization is little to none, which is not enough for the current purpose.

After the server-side logic execution performance has been optimized close to the maximum, the only ways to further improve the performance of the web application is to somehow increase the processing power of the servers or decrease the amount of required processing work per server. The traditional solutions have been the setting up of additional servers with a load balancer in front of them, investing to more powerful hardware for the current servers or some kind of combination of the previous two. As the goal is to avoid all of these options, other solutions are explored.

It is worth noting, that the problematic performance bottleneck in this kind of high traffic situation is the server, not the client. Therefore, this thesis will not pay much attention to the client-side logic optimization, although some of the server work could be moved client-side in order to lessen the workload of the servers. However, to really utilize this approach, the whole application architecture should be especially designed for this in

the first place, which is not the case with already existing applications.

1.1.2 Research questions

In order to find the most usable ways of improving the web application scalability without increasing the number or processing capacity of the servers, answers to the following questions are sought: What are all the options for lessening the workload of an origin server? Which of these options are currently utilized and at what rate? How could the situation be improved? What are the actual steps for making the improvements in practice? How does the application perform in each of these stages?

1.1.3 Research methods

In order to find the most thorough answers to the previous questions, a few specific research methods are utilized. First of all, the current status and utilization rate of different solutions to the scalability problem are discussed with the senior systems personnel of the company, as well as their suggestions for possible improvements. Secondly, the theory behind the systems personnel suggestions is studied alongside other closely related technology variants. Based on the studies, the most suitable improvement options for current needs are listed and put in order of execution. Before and after each concrete step of improvements, the performance of the application is measured.

The main goal of the thesis project is to improve the applications' scalability during high traffic peaks, thus making them more reliable under varying traffic conditions. This can be achieved by increasing the applications' ability to serve requests per time unit as much as possible. In other words, the goal is to increase the maximum **throughput** of the application.

The maximum throughput is measured first separately for the application front page, because it is the performance bottleneck of the original application. In addition to this, a more advanced simulation of a realistic high traffic peak situation is created in order to measure the maximum total throughput of the whole application.

1.1.4 Scope

The thesis is split into three major parts: the theoretical study, the empirical study and the analysis of the previous two. The theoretical study is conducted under the guidance of the company's senior systems personnel, as they have a lot of knowledge about the specific area of web application server performance and can point the study in the right direction from the beginning. As a result of the theoretical study, an ordered list of concrete application performance improvement steps is created alongside specific performance measuring criteria, in order to help during the empirical study.

The empirical study consists of a real life web application, in which the servers' ability to serve requests has been inadequate in the past during high traffic peaks, and the actual process of implementing the suggested performance improvements into it. In addition, the performance of the application is measured with appropriate tools at every step on the way, in order to keep track of the progression.

In the analysis part, the results from both studies are reviewed and analyzed. The advantages and disadvantages of the final optimized server application are compared to those of the alternative solutions, in order to understand the actual usability of the used approach. Additionally, the success of the thesis project is evaluated by determining if the desired performance improvement was reached or not and how thorough answers were found to the research questions set in the beginning.

1.2 Structure

The first chapter describes the actual research problem as well as the research questions and methods for answering them. Furthermore, it describes the structure, scope and special markings of the thesis.

The second chapter describes the basic principles of the Internet as a system: what is the relevant history behind it, how it is structured today and how it generally works on a higher level. A more thorough description about the part of the Internet that is closely linked to the specific research problem is presented. Furthermore, the current status of the Conmio web application pipeline all the way from the back end servers to the browser clients is introduced, containing the selected technologies, company conventions and the advantages and disadvantages of the current approaches. In addition, the web application performance is defined alongside common performance guidelines and measuring criteria.

The third chapter dives deeper into the actual solutions for the scalability issue; especially into the theory of caching and advanced caching techniques. It describes a few of the most promising caching techniques, Edge Side Includes, user group specific HTTP response caching and general HTTP response caching, in more detail.

The fourth chapter describes the implementation process of the Edge Side Includes technique into the application, as well as the performance improvements achieved with it. In the end of the chapter, the results are briefly analyzed to be good yet inadequate for the current purpose.

The fifth chapter describes the implementation process of the user specific HTTP response caching technique into the application as a further improvement alongside Edge Side Includes. The performance is measured with ApacheBench and additionally also with JMeter under a realistic high traffic peak simulation. Results of the measurements are briefly analyzed to be better than with plain Edge Side Includes solution, thus making this the number one improvement solution.

The sixth and final chapter of the thesis contains conclusions of all the findings made

during both theoretical and empirical studies conducted in the project. Further improvement suggestions are described and the reliability of the performance measurements is analyzed. Finally, the successfulness of the whole thesis project is reviewed and analyzed.

1.3 Stakeholders

In addition to the student and the school Tampere University of Technology (TUT), there is a third stakeholder in the thesis project: the company **Conmio Oy**, to whom the thesis is written for. The company has some special responsibilities regarding the thesis project and it will also benefit from the research conducted during the project.

The thesis is written for the company Conmio Oy in order to gain knowledge about the usability of Edge Side Includes markup language and other possible application scalability improvement solutions to be utilized in the company's future projects. The company is responsible for providing the student a suitable working gear and environment as well as separately agreed amount of time per week to be used for the progression of the thesis.

1.4 Special markings

Important terms that are explained also in the *Terms and definitions* table are written in **bold** when they appear in the text. Other important terms are written in *italics* in order to emphasize the special meaning of these terms.

2. STARTING POINT

2.1 Internet

In the developed countries of today, it is pretty common for people to have access to the Internet. In Finland, for example, the great majority of people have either used the Internet themselves or, at the very least, have heard about it. In these countries, the presence of the Internet can actually be so overwhelming that it is easy to forget how relatively new thing it is in the first place.

2.1.1 Beginning

The story of the Internet started back in the 1960s when an idea of using computers as a source for research and development in scientific and military areas was gaining popularity among researchers. This gave birth to the first version of the Internet, a system called ARPAnet, which first came online in 1966. It was named according to the Advanced Research Projects Agency (ARPA), which was also monitoring the usage of their new creation. As planned, in the beginning ARPAnet was mainly used for academic and scientific research purposes by the scientists and engineers of America's four main universities. [1]

During the second half of the 1970s, personal computers (PC) started to become more common, which had an effect on the development of the Internet. Programs like E-mail and Usenet became important tools of communication among individuals all over the world, leading to a situation where ARPAnet was no longer directed only towards scientists and engineers. Instead, it gained popularity also among common people especially because it consisted of systems that could be used independently for communication and information exchange. After a few successful decades, ARPAnet was shut down in 1990 in order to be replaced with its successor, the Internet. [1]

2.1.2 World Wide Web

One of the most important inventions of the Internet, especially in the scope of this thesis, is the World Wide Web (WWW), which is nowadays commonly referred as Web. It was invented and developed in the beginning of 1990s and released in 1991 by Tim Berners Lee with assistance from Robert Caillau. Lee's original idea was to have a fairly simple yet powerful enough Internet system, which would allow linking data together across

computers and networks. Lee designed the first web pages and used a web browser he had written himself, called WorldWideWeb, to view them. [1]

The real explosion in the popularity of the Internet came few years later when the first graphical web browser Mosaic was released in 1993 by the National Center for Supercomputer Applications (NCSA). As a consequence, web pages written in HyperText Markup Language (HTML) started to appear in the Web. Mosaic created a standard that was later mimicked by the more commonly known browsers Netscape Navigator in 1994 and Internet Explorer in 1995. All in all, the World Wide Web is the main reason the Internet became popular with everyone, as it is the part of the Internet the users can actually see. [1]

2.1.3 Traditional Web

The most common way of using the Internet has traditionally been the personal computer; it is the way it all started back in 1991. The first web pages contained mostly just pure text content. As the technology advanced, additional elements, like images, videos and audio, could gradually be added to the mix, thus allowing a richer user-experience. Despite these advancements in technology, the experience itself remained on the same platform for a long time.

Today, practically every web page consists of some version of HyperText Markup Language that binds different multimedia elements together with the help of closely related technologies like **server-side** programming language Java, **client-side** programming language JavaScript and output formatting language Cascading Style Sheets (CSS). [1]

While a lot of the Internet usage still happens through traditional personal computers, there is a big shift going on in the way people use the Web today. Rapid advancements in wireless technologies combined with a plethora of different mobile devices including smartphones, phablets, tablets, hybrids and laptops are changing the nature of the Internet. It is no longer enough to have a web site that looks nice on a desktop computer web browser; mobile web browsers have to be capable of showing the same content appropriately scaled for the available screen size. [2]

2.1.4 Mobile Web

The first software counted as a mobile web browser was the PocketWeb for the Apple Newton Personal Digital Assistant (PDA) created in The Telecooperation Office (TECO) in 1994. The first deployment of a so called microbrowser on a mobile phone happened in 1997, when Unwired Planet put their UP Browser on AT&T handsets to allow user access to Handheld Device Markup Language (HDML) content. [3; 4]

The critical step in mobile Web technology happened also during 1997, as an U.S. network operator Omnipoint was planning to release the first mobile Web service. They did

not, however, have any idea of how to actually achieve that, since there was no existing standard wireless Internet technology for mobile devices available at the moment. Therefore, an open competition for creating one was opened by Omnipoint. The competition received subscriptions from the four major companies of the time: Nokia, Ericsson, Motorola and Unwired Planet. While all proposed technology variants had their advantages, they all had also the big disadvantage of being proprietary solutions, which was not accepted by Omnipoint. In order to solve the problem, the four bidders got together and trashed out a standard. The result was the Wireless Application Protocol (WAP), which was especially designed to overcome the problems of the slow and unstable Internet connection; something the wireless technology inexorably was at the time. [5]

The first mobile phone containing WAP browser for Internet access was Nokia 7110, which was released in 1999. Though the browsing experience with the phone was still a far cry from the desktop counterpart, this was the first time a mobile phone could be used to view Internet content, namely Wireless Markup Language (WML), which was based on HDML and became the standard format for WAP pages. WML pages were usually stripped down versions of corresponding HTML pages and had to be separately created each time. Despite the downsides, WML remained the standard way of presenting mobile Web content for the first half decade of the 21st century. [6; 7]

Since the release of the first WAP browser in 1999, mobile web browser technology has always differed from the desktop counterpart. Although the Wireless Markup Language used in WAP is quite similar to the HyperText Markup Language used in WWW, they are still two different standards. In addition to this and the obvious mobile device limitations, like smaller screen size and slower data connection, much of the difference has become from the usage of the less advanced technology variants in all mobile client-side technologies, including simplified version of JavaScript, called WMLScript, and CSS, namely Wireless CSS (WCSS). [7; 8]

2.1.5 Responsive Web

During the last five years, both microprocessor and mobile data connection technologies have taken huge leaps in advancement, which has lead to a great increase in the mobile device processing power and overall user experience; the mobile phones of today are like small computers.

In addition to the hardware, mobile web browsers have also advanced towards the desktop counterparts and are actually using mostly the same standards already. This means that WWW has largely replaced WAP and made it almost useless especially in the more developed countries. Furthermore, a vast array of end devices between mobile phone and personal computer has arisen, which means almost unlimited number of different screen size-feature set -combinations. Instead of creating a separate version of a web site for each possible variant, responsive design is aiming to create one universal version of the

site; one that will adjust the content according to the available screen size and resolution as well as other available features. [9]

Since responsive design aims to make it possible to create the web site once and then browse it with all end devices, it is pretty likely that it is the way of the future in all web design, although most of the time neither the technology nor the media companies are quite there yet. However, as making a web site as responsive as possible has many advantages, it definitely affects to the implementation already today.

2.1.6 Second screen applications

The advancement of the web technologies have improved the Web experience in many ways over the years by making it more accessible, interactive, functional and pretty. In addition to these, some totally new ways of utilizing the Web have been enabled by the new end devices. These include web applications that are utilizing the *geologation information* available via GPS in most modern smartphones, device *orientation and movement data* available via integrated gyroscope and just the bare *mobility* of the device.

One recently emerged group of web applications utilizing one or more of the previous possibilities is the so-called **second screen** applications. The basic idea of these applications lies in the nature of the mobile devices, as they are often used also while watching television; hence the name second screen. The usual purpose is to allow some kind of interaction with a specific television show in one way or another. Popular ways of interacting include *answering to a poll*, *voting a certain participant* and *chatting with other watchers of the show* for example via integrated Twitter channel.

2.1.7 Scalability

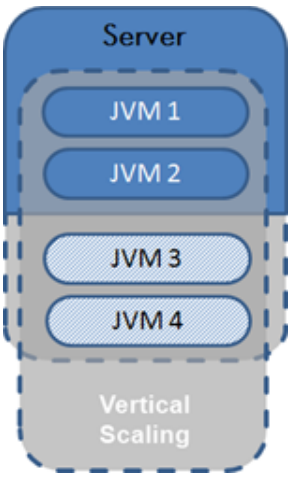

Figure 2.1: Vertical scaling by adding hardware resources on to existing servers. [10]

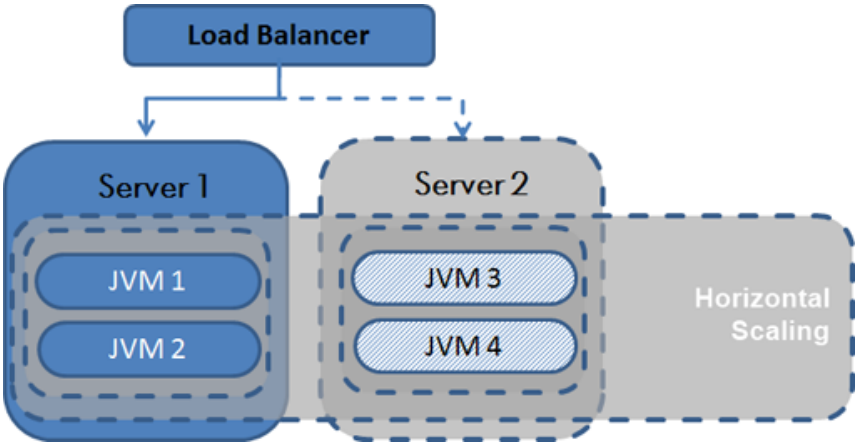

Figure 2.2: Horizontal scaling by increasing the number of servers. [10]

The popularity of the Internet has been growing almost exponentially since the release of the first graphical browser in 1993. This is due to the improvements in technology and availability of the Internet all over the world, in which mobile devices and mobile Internet connections have arguably been an important factor. [11]

The growth in the popularity of the Internet equals to more clients to be served, which then increases the load of the servers, leading to the need of increasing the performance of a single server or deploying more servers in turn. This means that the service is being **scaled** either vertically (**vertical scaling**, Figure 2.1) or horizontally (**horizontal scaling**, Figure 2.2). While both of these options are reasonable up to a certain point, things become unnecessarily difficult especially in cases where the server load differs greatly over time. In other words, there is an issue in the server **scalability**, which describes the system's ability to adjust its resources according to the current load. [10; 12]

A good example of this kind of situation has become tangible along with the second screen applications, as they are often tied to the air time of a certain TV show. Whereas the load of the web servers may be minimal for the major part of the week, it grows exponentially during the time the show is on, which usually means one to two hours per week. Even more challenging is the situation where during the show there is a shorter time frame to do some specific activity, for example to vote a certain participant in a poll. This may lead to a situation, in which a great number of watchers access the application during the same 5 minute time frame, causing an HTTP request overload on the servers and inability to serve all clients. To the users this is shown as either unnecessary long loading times or total *page not found* responses. In other words, the application is broken from the user's point of view, which is never the desired outcome.

2.2 The basic structure of the Internet

Although the history of the Internet itself is still fairly short, it has the nature of connecting together various protocols, machines and technologies, many of them having prior existence elsewhere. When even the tiniest advancements in these individual areas of the Internet are combined, the overall advancement of the Internet as a system has become very rapid. Because of these reasons, much of the Internet-related technology is not very well standardized and also the existing standards are frequently updated. This can be clearly seen for example with the constant emergence of new programming languages and frameworks. [13]

However, some specific areas of the Internet are in turn pretty strictly standardized. One of these areas is the basic structure of the Internet. The original idea of the Internet being a distributed system that links together computers and networks strongly affected to the creation and selection of protocols it was built on top of, as they once thoroughly fulfilled the needs of what the Internet originally was; a worldwide network of clients and servers, in which one of the former would **request** a static document from one of the latter, which would then comply and return the requested document as a **response**. The core protocols for handling these requests include the protocol for transferring hypertext data, called **HyperText Transfer Protocol (HTTP)**, as well as protocols for handling the data transferring in more general level, namely **Transmission Control Protocol (TCP)** and **Internet Protocol (IP)**, commonly working collectively as **TCP/IP**. [14]

Although much has changed after the birth of the Internet in 1990, these core mechanics have largely maintained their position as the basic building blocks of the Internet. On one hand, they have allowed the Internet to grow and evolve over the years to the current state, but on the other hand, they have also presented restrictions and challenges that the web developers have somehow had to bypass. [14]

2.2.1 From web pages to web applications

Internet originally consisted of separate static hypertext documents that each represented one web page. These pages could be cross-referenced through hypertext links and users could navigate between them, but the pages would not necessarily have anything in common; no similar theme or look. This kind of separate web pages are quite rare in modern Internet. [14] However, it should be noted that the definition of the term has since extended to mean all non-embedded resources obtained from a single URI using HTTP. [15]

The more advanced version of a web service is called a web site, which is also probably the most common term for the end user when talking about an Internet service. A web site contains a set of connected web pages, each of them corresponding to a common theme and look. Usually a web site contains also a more refined navigation system located for

example in the top part of every page as a navigation bar. The biggest limitation of pure web site approach is that the individual pages have to be static, yet this is also the biggest advantage of this approach, as the site then contains only the minimal amount of code and functionality. For some purposes a set of static pages is still enough, which makes it a considerable option even today. [14]

The most advanced version of a web service is called a **web application**. It can look similar to a common web site in many ways, but the difference is that the pages are no longer just static content. Instead, the page content can be dynamically tailored for each user within each request-response interaction. This usually requires that the application state information for each user is stored somewhere between requests and is available when rendering the response. [14] Web application is also the term used in this thesis to describe the kind of web service a second screen application represents.

The great majority of today's web sites are actually web applications, as it allows a richer user-experience and customized content, which are things most are striving for. At a technical level this means that some kind of data processing has to happen at some point between the initiation of the original request and the displaying of the final response page to the user. When this happens inside a pure client-server model, it means the required processing can be done either server-side or client-side, yet most of the time it happens partly on both ends. [14]

2.2.2 Client-server model

A pure client-server network is a **two-tier** software architecture, which is currently the most commonly utilized form of distributed network computing. In the beginning, it was only necessary for the servers to run a server program that would listen to connections from clients, and when one appeared, the server should find and return the requested resource to the calling client. These interactions were separate from each other, with no state information saved between requests. [16]

The server has classically been responsible for most of the processing tasks of a request-response interaction. However, as the time has gone by, many new ways of utilizing the Internet have emerged, and the core mechanisms of Internet have not always totally satisfied the needs of these new applications. Therefore, various new mechanisms have been invented to extend the core Internet functionality. These include the mechanism for requesting only a fragment of a document from the server, called **Asynchronous JavaScript And XML (AJAX)**, as well as mechanism for storing the state information between requests, which is usually achieved with session variables, cookies, databases or combination of the previous. Moreover, client-side processing has become more common and even the dominant way of doing processing in many cases, which has had an effect on the overall architecture of these applications. [16]

2.2.3 Server-side architecture

The term **server** can refer to a particular application, program or software module that is capable of performing computing upon request. It can also refer to the hardware platform or appliance that runs one or more of the previous. Considering the client-server model, the basic principle has classically been that the servers can advertise their services to the clients, but the servers do not send any data to the clients without a request. [16]

On a more technical level, a server usually refers to a set of software components that work seamlessly together in order to provide suitable services to the requesting clients. In web application world, this usually means that a physical server runs a **web server** software, which is connected at least to an **application server** software with potential connections to other kind of server software, including database servers, file servers and mail servers. The web server is usually the initial recipient of clients' requests, which then interprets and forwards the requests to the corresponding server programs, usually to the application server. An example server configuration is illustrated in the Figure 2.3

It is worth noting that each of these server programs can run either in the same physical server, each of them may have their own physical servers or the configuration can be something in-between; the final configuration is decided based on the requirements of a particular web application. However, as server hardware optimization is not really the topic of this thesis, only minimal amount of attention is paid to the server hardware while the software will get more thorough inspection. [17]

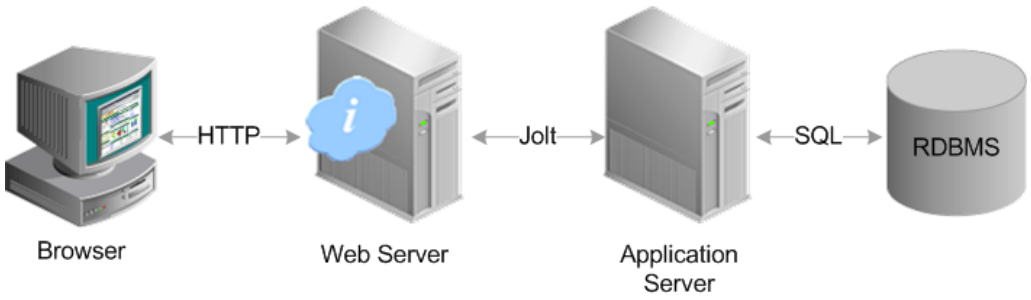

Figure 2.3: An example of a server configuration where all different server programs reside on separate machines. [18]

2.2.4 Server-side programming

Regardless of the selected server configuration settings, there is a variety of scripting languages that can be utilized when creating the desired server-side functionality for a web application. According to W3Techs statistics, currently the most popular server-side scripting languages are PHP, ASP.NET and Java. [19]

PHP originally stood for *Personal Home Page (Tools)*, but is now said to be a "recursive acronym" for *Hypertext Preprocessor*. [20] Whatever the case, the name already implies for what PHP was originally designed for: creation of dynamic web content. When the original purpose is considered alongside the easy learnability, flexibility and cross-platform support of PHP, it is no real surprise that currently 82% of all web applications are powered with it. [19; 20]

However, while the flexibility of a programming language is clearly beneficial, it can also be one of its greatest downsides. This is the case with PHP, as the code written with it can easily turn into spaghetti if no clear programming conventions are followed. PHP is also less efficient and scalable than some of its counterparts. [21; 22]

ASP.NET is not actually a language of its own but a web framework from Microsoft for building web sites, applications and services based on the popular .NET Framework technology. The goal of ASP.NET is to make web programming easy and similar to normal application development, for example by allowing developers to use any programming language supported by the .NET Framework when writing server-side code. [23]

While being a reasonable choice for many companies by offering a variety of tools and support, ASP.NET has the downside of being a commercial product by Microsoft. Therefore, companies using ASP.NET are tied to Microsoft in many ways: ASP.NET is designed to run on Windows platform, which means Windows servers have to be used, programming has to be done usually with Visual Studio and additional licensing costs have to be paid to Microsoft with each new server and load balancer. [24]

Java is an object-oriented programming language originally designed for stand-alone application development, but quickly expanding also into other areas of development; most importantly web development. The main goal of Java is to make it possible to write the code once and run it everywhere. To make this happen, Java programs are usually first compiled into Java bytecode and then run in **Java Virtual Machine (JVM)**, thus making the code platform-independent. [25]

To the date, Java has become one of the most popular programming languages with great community support, documentation and compatibility with different platforms. With the additional features of **Java EE** platform, like **Servlets** and JavaServer Pages (JSP), the development of dynamic web applications has been abstracted into Java classes, thus making web application development well-structured and standardized. Moreover, as server-side functionality built on Java tends to be more efficient and scalable than that of its counterparts, it is commonly the preferred technology especially in high traffic web sites and applications. [26; 27]

Though being a good choice in many ways, Java is not without its shortcomings. Compared to PHP, for example, Java is considerably more difficult to approach for inexperienced developers, which can make it an overkill especially if the project is relatively small. The application development process itself is also much heavier with pure Java,

as the code has to be constantly compiled and deployed in order to see the results of the changes, which inevitably slows the development process down. However, many of these problems are addressed in multiple Java-based programming frameworks that are available today. [28]

2.2.5 Client-side architecture

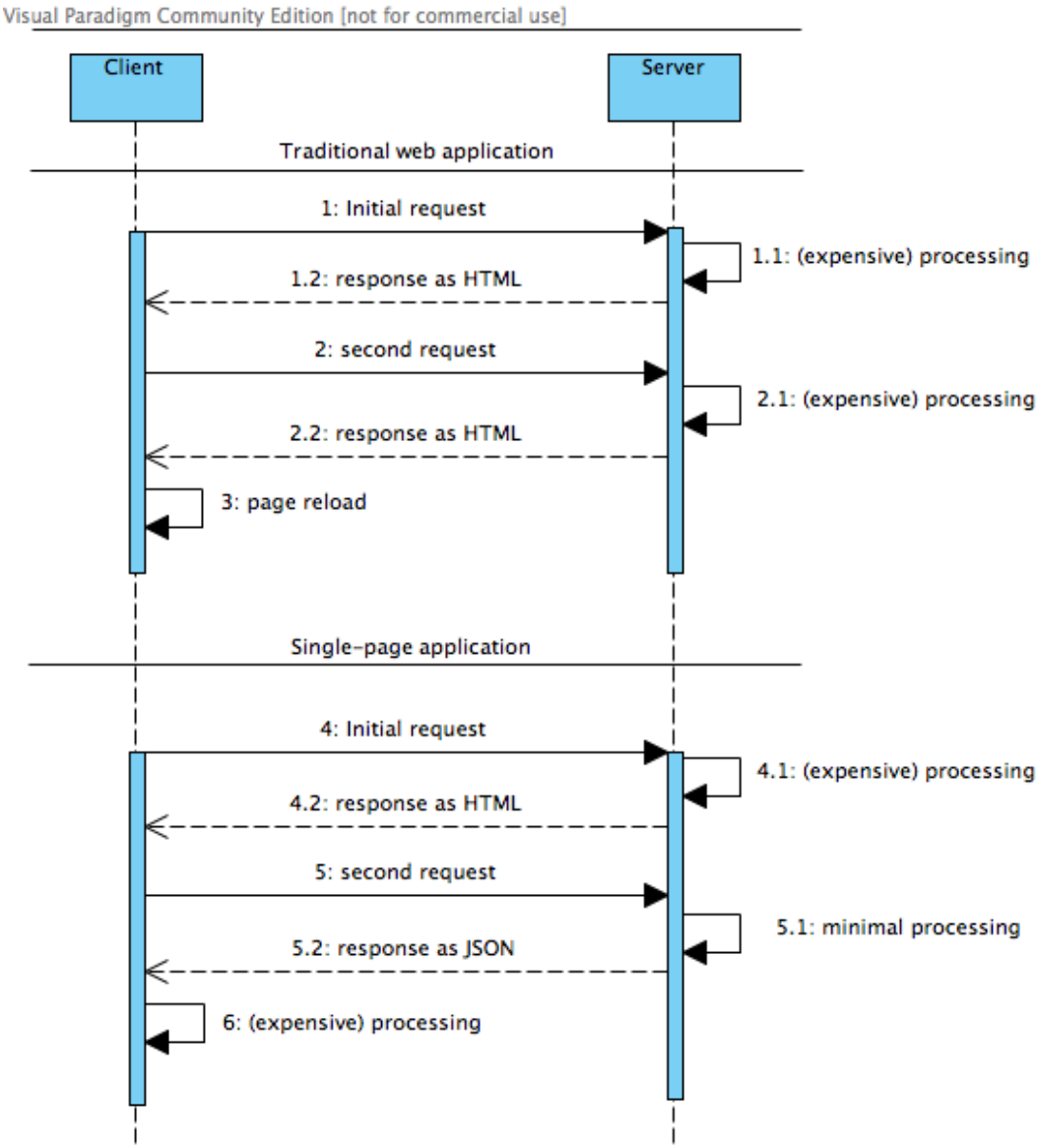

Figure 2.4: The difference between traditional and single-page applications.

The term **client** refers to the requesting party of a request-response interaction. Inside the client-server model, the client is nowadays most often a web browser, though it can be also any other program sending requests to servers through a data connection. Historically, the client-side architecture has advanced from terminals to thick clients, from there to thin

clients and finally to a special breed of clients, web browser clients. [29]

Terminals aside, the term thick client refers to the first desktop systems of the 1980s that were equipped with keyboards, mouses and software applications. These systems could do various things client-side, including data validation, graphical processing and even some of the business logic processing, but they had very limited amount of processing power. [29]

The natural progression was that most of the processing tasks were pushed to the servers, as they possessed much more processing power than the client systems. This resulted in thin clients that were designed to be lightweight applications only capable of doing the simplest processing tasks, like input validation, while leaving the more complicated business logic processing tasks for the servers. [29]

Along with the birth of the Internet came also the web browsers, that have since become the standard type of client software applications. Web browsers standardized the HTTP protocol and various related things, like the markup language for web documents, namely HTML. [29]

In the beginning, web browsers were much like thin clients, performing only simple text and image processing tasks client-side. However, as the client-side technology continues to evolve, increasing amount of more advanced processing tasks can be done client-side nowadays. This is the current trend in web application development; in a way, the development is going backwards, as thin clients are becoming thick clients again. However, this time the client-side hardware is much more powerful, and has usually access to various kind of additional data, like user's location. [30]

Web applications that do most of their processing client-side are called **single-page applications (SPA)**. The most important difference between a traditional and a single-page application is that in the latter, a full page load is required only once when opening the application, while the following view updates are done asynchronously through AJAX. In practice, this means more complex logic is required on client's end, as the responses have to be parsed and interpreted before the desired HTML can be viewed to the user. The difference is illustrated in the Figure 2.4. [31]

2.2.6 Client-side programming

The movement towards client-side driven processing inevitably increases also the amount of required client-side code. The most common client-side programming language currently by far is JavaScript, which is used by 87,8% of all web sites, and the percentage is continuously increasing [19]. Other languages, like Flash and Java were previously used in client-side scripting, but their popularity has been decreasing, as JavaScript has become the standard language of client-side programming. [19]

Despite the supreme popularity of JavaScript, the language itself is quite far from supremacy. As a programming language, JavaScript definitely has many good features,

like simplicity and object prototypes, even though it was originally written in just 10 days. However, the most obvious disadvantage of JavaScript is that the developer cannot control the client's runtime environment, as it depends on the browser. In addition to this, the language itself may behave inconsistently in some situations due to the inconsistencies in the language specifications. [32]

Though selecting the client-side programming language is easy, selecting the most suitable **front end** frameworks and libraries can be a bit more daunting task. For a long time, it has been common that client-side scripting has been done either with pure JavaScript or with the help of some popular JavaScript library, like jQuery. However, when complex enough functionality has to be created client-side, like with single-page applications, using only a single library can lead to messy code. To address this issue, a number of front end frameworks have been created.

While reducing the server workload is the purpose of this thesis, and though it can be achieved to some extent by moving the work client-side, it is not the most optimal solution. For one, the front end technologies have not yet standardized, as can be seen from the sheer number of frameworks and libraries solving the same problems. [19] Secondly, though mobile web browsers have advanced a great deal over the last few years, they are still behind desktop counterparts in some areas, which means not everything will work in a mobile web browser. Third, practice has shown that the development process of single-page applications is slower than that of more server-oriented applications. This is understandable, as client-side processing functionality should be tested in theory with all supported end devices, while the same functionality done server-side has to be tested only once. Finally, switching to a single-page application design would not solve the problems with existing applications without heavy rewriting of the code and architecture.

2.2.7 Multi-tier architecture

Even though most of today's distributed networking systems are based on client-server model, which in its purest form is a two-tier network, many real life systems are actually **multi-tier** networks, meaning that between the client and server lies some kind of **middleware layer** negotiating the transactions between the two. In practice this means that instead of the web **traffic** going straight from the client to the server and vice versa, it goes through one or more **proxy servers** on the way. These proxy servers can be used for various purposes, like content caching, content aggregation or load balancing. The difference between 2-tier and 3-tier architectures is illustrated in the Figure 2.5. [16]

From the developer's point of view, it does not usually matter whether there is middleware layer in the play or not, as long as everything is configured properly. However, from the client's perspective, conveniently configured middleware with suitable features can offer tremendous boosts in perceived application performance. This is also the key for achieving the desired performance boost that is the main goal of this thesis.

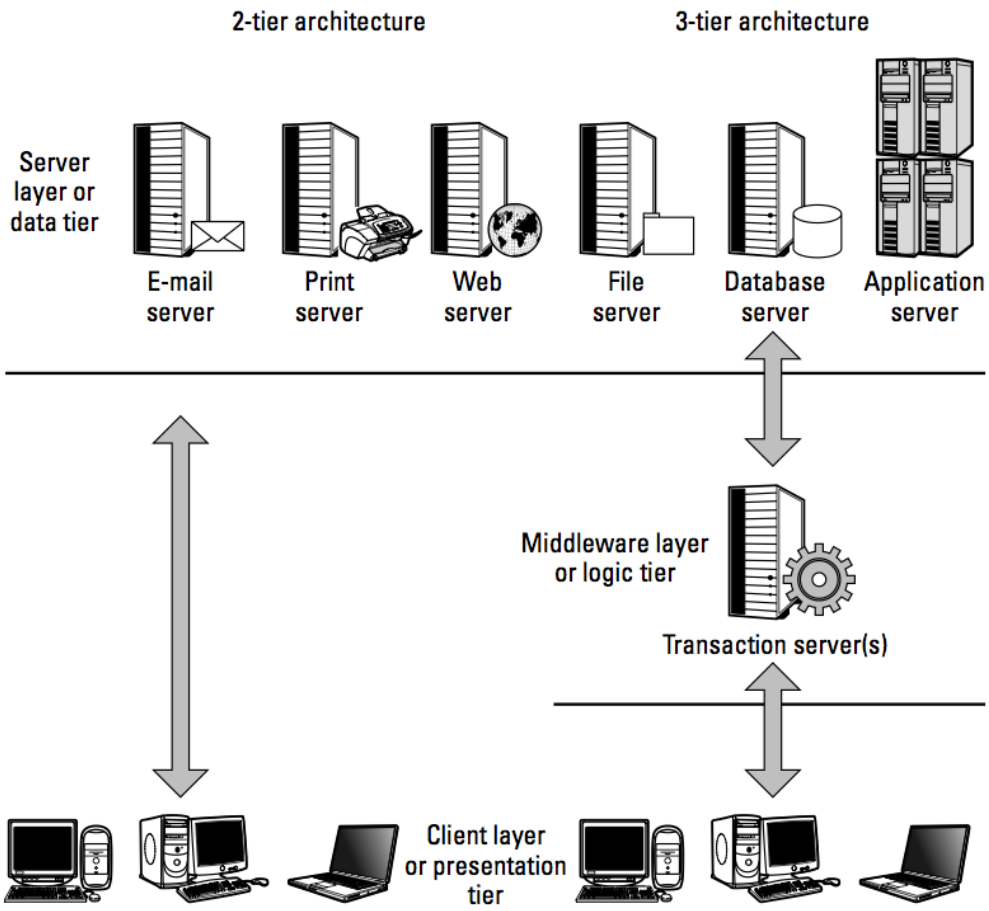

Figure 2.5: The difference between 2-tier and 3-tier architectures. [16]

2.2.8 Conmio web applications

Since its birth in 2002, Conmio has always been a software company focusing mainly on the development of highly usable mobile web sites and applications. As the time has gone by, the interactivity and the need for customized content in applications have become more important factors, which have had an effect on the architecture of the applications.

From the end user's point of view, web applications created by Conmio have usually been offering value especially for the news- and sports-hungry users on the go. However, many other types of applications have also been created during the years, including search, chat and most recently second screen applications.

2.2.9 Conmio web application architecture

Conmio web application architecture is based on proven open standards as much as possible. Most of the applications are built on top of Java Virtual Machine (JVM) with **Apache HTTP server** working as a web server and **Apache Tomcat** working as a **servlet container**, which is kind of an application server for the Java platform containing the actual

application. In addition to these, load balancers and other proxy servers, like Varnish and **Content Delivery Networks (CDN)** have been utilized especially in the architecture of the more popular web applications of the company. An example of a Conmio web application architecture is illustrated in the Figure 2.6.

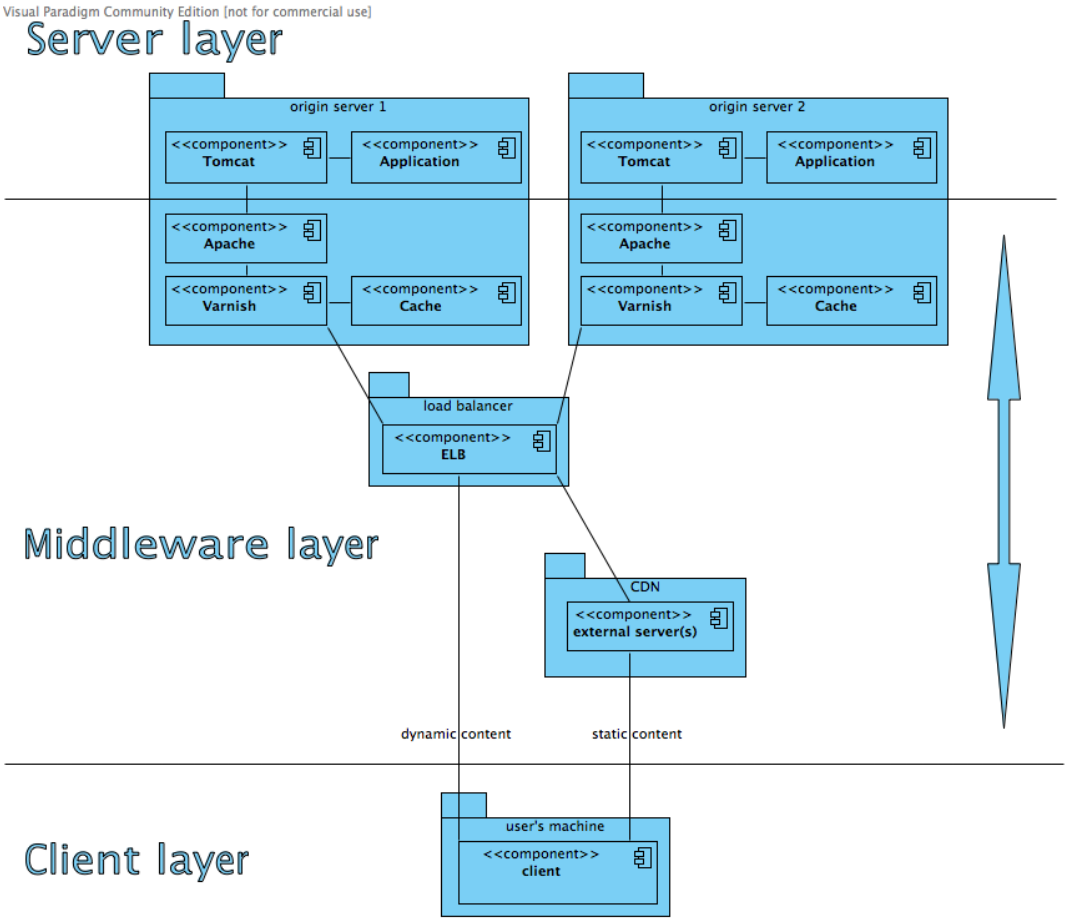

Figure 2.6: Example of a Conmio web application architecture.

From the developer's perpective, a classical Conmio web application is a working compromise between familiar standards, like Java, Tomcat and JavaScript, and state of the art technology, including HTML5, LESS and Node.js. In addition to these, Conmio has used the Grails framework as the heart of most of the applications during the last few years. Even though newer frameworks have emerged, Grails still seems to offer the best overall productivity in most of the cases, especially if the project is relatively large.

2.3 Programming frameworks

A programming **framework** is a set of source code or libraries that provide functionality common to a whole class of applications.

2.3.1 Purpose

The main purpose of a programming framework is to provide specialized tools for application developers for reducing the need of writing repetitive code, thus allowing them to focus more on the **business logic** itself. Compared to libraries that usually provide one specialized functionality, frameworks provide a broader range of functionalities that all are used by certain type of applications. [33]

The most relevant example of a programming framework in the scope of this thesis is a **web application framework**. Web application frameworks provide specialized functionality concerning common web application feature requirements, like user session management, templating, security and data storage. [33]

Different frameworks offer different degree of flexibility versus features, which makes certain frameworks more suitable to certain projects than others. It is often left to the developer to choose which framework to use in which project, but the choice can be based also on company's conventions or other leading forces. [33]

2.3.2 Advantages

Using a programming framework in a project offers various advantages, provided that the selected framework is suitable for the current project. Like web applications, many application types usually have a lot of common functionality underneath, even though the look and feel of an application can be totally different from another. Therefore, creating the same core functionality all over again would make the development of applications repetitive in the long run, which is why frameworks encapsulate a lot of these otherwise repetitive tasks into features that make things easier for the developers. This leads to greater developer productivity and happiness, better-formed and more standardized code as well as generally more robust, error-free and secure applications. [33]

Frameworks usually provide also tools for splitting the code into reusable modules, often called plugins. This makes it easier to split development tasks for those who program the framework, meaning the plugins, and those who program the final application. Moreover, frameworks usually direct developers to follow a certain design pattern and related best practices, which unifies the development process even more. [33]

2.3.3 Disadvantages

While using a suitable programming framework in a project definitely offers many advantages, those usually come with a price. The most common downside of using a programming framework is the downgrading of the application performance. This is usually caused by the generalized code, that has to make additional checks for different scenarios in order to determine the correct path of action in each situation. [33]

Moreover, the learning curve of using a programming framework efficiently can be relatively high, which means a fair amount of studying and learning for each developer. Therefore, using the same framework repeatedly is usually more efficient than switching it on a project by project basis. [33]

Even though frameworks usually make applications more secure, occasionally there can be bugs in the framework code itself, which can create a security threat for each application using the particular framework. However, this kind of situation is pretty rare and usually swiftly patched. [33]

2.4 Grails

Grails is a web application framework that has been used as a heart of most of the Conmio web applications in the last few years.

2.4.1 Main features

Grails is an Open Source, full-stack web application framework built on top of Java Virtual Machine (JVM). The main features of Grails include the utilization of **Groovy** programming language, Convention over Configuration (CoC) paradigm, Spring framework, Hibernate framework and Sitemesh framework. [34]

Groovy claims to be an agile and dynamic programming language for the Java Virtual Machine, building on top of Java strengths but with additional power features inspired by other programming languages, like Python, Ruby and Smalltalk. This means that most of the valid Java code is also valid Groovy, and classes written with one of the languages can be used interchangeably. [35]

Utilization of Convention over Configuration paradigm basically means providing reasonable default configuration values for the project. This way the developer only has to configure the unconventional aspects of the application, which greatly reduces the amount of work in most cases. [36]

Spring framework is a popular Inversion of Control (IoC) container for the Java platform. It is designed to help with the infrastructure of a Java web application, by providing many useful features to help with things like handling object lifecycles of specific objects and injecting functionality to other classes through dependency injection. [37]

Hibernate is an Object-Relational Mapping (ORM) library for Java platform. It provides a framework for mapping an object-oriented **domain model** to a traditional relational database. The Grails implementation of this is called Grails Object-Relational Mapping (GORM). [38]

Sitemesh is a lightweight layout and decoration framework for Java web applications. It allows a clean separation of content from the presentation. [38]

2.4.2 Plugins

Plugins are Grails standard for creating reusable functionality to be used in various projects. The Grails community has created a plethora of publicly available plugins to help with many standard features, like sending e-mail from the server. In addition to that, there are many Conmio-specific plugins available for all the company projects, helping with things like content retrieving and caching. A few specific plugins created by Conmio, especially in the scope of this thesis, are related to device recognition, content caching and content modularization. The first one is called **Conmio Devices**, the second **Conmio Cache** and the third **Conmio Modules**.

Conmio Devices has a long history at Conmio, as device recognition played a major role especially in the past when implementing web applications, as the capabilities of end devices differed more from each other than today. The main purpose of the plugin is to offer an unified way of retrieving information about the current end device with some additional features, like enabling device grouping based on certain criteria.

Conmio Cache has also a fairly long history at Conmio. The plugin's main purpose is to easily allow the caching of Groovy and Java objects as well as markup fragments, which greatly reduces the amount of required **back end** processing when the expensive processing results are retrieved from cache instead of executing the same processing tasks all over again with each request.

Conmio Modules is a bit more recent creation of the company. It was created to allow the modularization of a specific markup fragment and the required logic for composing that markup in the first place. Examples of these modules include menus, news listings and social discussion feeds.

2.4.3 Advantages

The main goal of Grails has always been to drastically simplify enterprise Java web development. This is achieved through diminishing the amount of required configuration and by utilizing other well-known technologies already familiar to many Java web developers, such as Spring and Hibernate. Furthermore, Groovy is built on top of Java, which makes it easier for the Java developers to start writing Groovy code, since almost all valid Java is also valid Groovy. [38]

From the developer's point of view, writing Groovy instead of Java is often a pleasurable experience, since it makes things more flexible, but only if the developer wants so. The additional features Groovy provides makes it possible to achieve similar functionality with considerably smaller amount of code than with pure Java. [39]

As stated before, Grails has been used as the heart of many Conmio projects in the last few years. Therefore, the employees of the company have gained a lot of knowledge and experience regarding the usage of Grails, which has further enhanced the developer

productivity in projects utilizing Grails.

2.4.4 Disadvantages

Like with most of the programming frameworks, the biggest downside in using Grails is the deterioration in the application performance. According to TechEmpower web application framework performance comparison statistics, Grails is positioned around the middle of all frameworks in terms of performance. [40]

Even though Grails utilizes Convention over Configuration paradigm, it is still built on top of Java and Spring, which equals to a pretty large infrastructure. In some smaller projects this kind of infrastructure can be too much and the project would benefit from using a lighter framework.

2.4.5 Alternative frameworks

In addition to Grails, there are a lot of alternative web application frameworks available, of which some of the most interesting ones lately have been Node.js and Angular.js.

Node.js is actually a software platform for scalable server-side and networking applications, meaning it is directed towards the back end of the web application development. It supports things like HTTP and socket communication, which allows it to work as a web server without additional server software, like Apache HTTP server. Node.js applications are written in JavaScript. [41]

Angular.js is designed to help especially with the front end programming of single-page applications. It works by replacing special attributes on the site with specified functionality. Like Node.js, Angular.js is based on JavaScript. [42]

Both of these frameworks have a lot of good features that could make them useful in many projects. However, out of these two only Node.js could actually be considered as a replacement for Grails, since it is directed towards the back end of the applications. In fact, Node.js has already been used in some of the smaller projects of Conmio, but they have proven that neither the framework nor the company conventions are ready yet for switching totally from Grails to Node.js. Part of this is because of the inferior features of Node.js compared to Java, like logging and security, and the other part is the amount of knowledge about each framework inside the company. All in all, Grails still seems to offer the best value for most of the Conmio projects, and therefore it stays as the number one choice in the company projects, at least for now.

2.5 Web application performance

Over the last decade, the complexity of web applications has increased dramatically. While the applications' features themselves are more complex than ever before, also the number of requests to be fulfilled is constantly increasing.

Rule Number	Description
1	Make fewer HTTP requests
2	Use a content delivery network
3	Add an Expires header
4	Compress components with Gzip
5	Put CSS at the top
6	Move JavaScript to the bottom
7	Avoid CSS expressions
8	Make JavaScript and CSS external
9	Reduce DNS lookups
10	Minify JavaScript
11	Avoid redirects
12	Remove duplicate scripts
13	Turn off ETags
14	Make AJAX cacheable and small

Figure 2.7: Rules for faster front end performance by Steve Souders. [43]

2.5.1 Definition

The web application performance can mean the efficiency of many aspects of an application. In the scope of this thesis, the most relevant aspect of a web application performance is the maximum throughput of the application, meaning the *maximum number of requests per time unit the application is able to serve*.

2.5.2 Performance best practices

Even though the aim of this thesis is to improve the back end performance of a web application, from the end user's point of view, the effect is verifiable on front end side, meaning the client. Therefore, it should be first made sure that the front end is not the performance bottleneck.

In 2007, the chief of performance at Yahoo!, Steve Souders, created a set of 14 rules for faster front end performance. Those rules have since become widely accepted standard performance guidelines in all web development of today. The rules can be seen in the

Figure 2.7 [43]

Based on these guidelines, a set of general best practices for improving web application performance has been formed. These best practices are based on compression, caching, minifying and bundling of static files, HTML optimization, image optimization, ETags and Content Delivery Networks. [44]

Compression is an algorithm that is used to remove unwanted redundancy from a file in order to reduce the size of the file. Compression can be applied to almost any kind of files returned via HTTP, thus making the amount of transferred data considerably smaller that it would otherwise be. [44]

Caching on its most basic level means HTTP caching, in which specific headers are added to the HTTP responses, telling to the browser that the particular content does not have to be reloaded from the server with every request. Instead, the version found from the browser cache can be used until the specified expiration time has been reached, leading to fewer HTTP requests and smaller amounts of data to be transferred. [44]

Minifying and bundling of static files are on par with the compression, since the purpose of minification is to remove unnecessary characters and spaces especially from the JavaScript files while maintaining the original functionality. Bundling means combining multiple JavaScript and CSS files into one of each kind, again leading to fewer HTTP requests. [44]

HTML optimization basically means the correct positioning of CSS and JavaScript files on a web page, removal of duplicate scripts and utilization of HTML5 optimization features when possible, all leading to more optimal rendering pipeline. [44]

Images are a huge part of today's web applications, meaning that reducing the size of image files can lead to tremendously smaller data amounts to be transferred. Image compression can be either lossless or lossy, former being less efficient but not affecting to the quality of the original image and latter being more effective but resulting to worse quality result image. [44]

ETags are headers containing unique strings, that can be used to validate the contents of a browser cache. This might seem like an useful feature, but like the Sounders' guidelines suggest, these tags should not be used but removed completely instead. The reason for this is that most modern web applications are behind a load balancer, meaning that each request could be served by a different server, which would lead to cache misses for identical content, because each server would generate a different ETag for the same content. Instead of making the number of HTTP requests smaller, this would actually increase the number of requests, which is not the desired outcome. [44]

Content Delivery Networks are a collection of servers located around the world, that contain a clone of a web application's static files. This way, all of the requests do not have to go all the way to the origin server but they can be served from the CDN servers instead. In addition to lessening the number of requests coming to the origin servers, the CDN

servers are usually much closer to the client geographically, leading to faster response times, therefore improving the performance of the application. [44]

2.5.3 Load balancing

After utilizing the previous best practices for better web application performance in order to achieve the best possible vertical scalability for an application, the performance can be further improved through horizontal scaling. This means increasing the number of origin servers and placing a load balancer in front of them. In practice this means a proxy server configured to receive all incoming requests, forwarding them to one of the origin servers for processing based on certain logic and then returning the response back to the calling client. [45]

In theory, every added origin server could multiply the maximum throughput value of the web application by the corresponding value of a single origin server. In practice, however, this is usually not happening and every added server will increase the maximum throughput value less than the previous server. This is often due to the lack of centralized caching system and the required calculations for all forwarding operations. [45] In addition to increasing the application performance, using a load balancer is also an useful way to increase the application stability, since in case of an origin server breakdown, there are still other servers able to serve the client.

2.5.4 Conmio web application performance

At Conmio, all of the previously described web application performance best practices have been utilized at least to some extent. There are many Grails features or plugins, created either by the Grails community or by the company, that are taking care of compressing, minifying and bundling of the CSS and JavaScript files, placing them at suitable locations on a page, adding suitable cache headers to the responses, compressing and caching the images and other content as well as utilization of Content Delivery Networks for serving static content. [38] In addition to these, load balancers are utilized in most of the applications by usually balancing the load between 2 to 8 origin servers.

2.5.5 Measuring the performance

In order to make comparisons with web applications utilizing different degree of optimization techniques, it is crucial to be able to measure the performance of those applications somehow. As defined before, the most important meter of performance in the scope of this thesis is the application's maximum throughput value.

ApacheBench is a single-threaded command line tool originally designed for measuring the performance of an Apache HTTP server, though being generic enough for testing the performance of any web server. Like Apache HTTP server itself, the tool is open

source, comes bundled with the standard Apache source distribution and is capable of measuring just what is required; the number of served requests per second per page. This makes it an ideal tool for measuring the web application throughput performance in the scope of this thesis. [46]

JMeter is another load testing tool by Apache Software Foundation. It is more advanced than ApacheBench, enabling the creation of more realistic simulations concerning the origin server traffic, thus allowing the measurement of the maximum total throughput of the whole application. [47]

In order to achieve the most reliable results, a certain measurement configuration is set up. To a computer containing hardware similar to the actual server machines, meaning 4 gigabytes of RAM and CPU with two cores, is installed nothing but CentOS, Java, Apache Tomcat, Apache HTTP server and Varnish. The computer is then configured to act as a **dedicated server** for the measurements. In theory, the student's computer should act as the client, but as the network connection speed could become the performance bottleneck in this setting, the ApacheBench instance is also run on the same machine, thus revealing the actual maximum throughput of the application more accurately. This way, the only things affecting to the application performance are the usage of Varnish and the application itself. However, JMeter is a more complex application written in Java, which makes it sensible to run it from the student's computer, so that it will not cause any additional load to the test machine.

The first few sets of benchmark results are dismissed with every particular measurement configuration in order to allow the normalization of the results, since both the Java Virtual Machine and the caches need some time to warm-up before they perform in the optimal level. The throughput values presented in this thesis have the margin error of 5%, meaning the actual value is somewhere in the margin of the *measured value* $\pm$ *5%*.

3. IMPROVING THE SCALABILITY

3.1 Caching

Out of the web application performance best practices described in the previous chapter, the most useful method in the scope of this thesis seems to be **caching**. This is due to the almost endless possibilities of using caches at different levels of the web application pipeline.

3.1.1 Theory

The term **cache** comes originally from French and means, literally, *to store*. In data computing, caching means storing of recently formed computer information for future reference, which may or may not be used again. Caching is useful only when the cost of storing the information is less than computing the same information again. This means it is especially useful to store information that is accessed frequently and the computing process of that information is **expensive** in terms of resources like processing power and time. [48]

The cache usefulness can be measured in terms of cache hits and misses. The requested information is searched from the cache based on the **cache key**, which is an unique value that can be linked to only one **cache entry**. Whenever the requested information is found from the cache, it is called a **cache hit**. Similarly, when the requested information cannot be found from the cache, it is called a **cache miss**. The percentage of all hits out of all requests is called the **cache hit ratio**. The performance improvement a cache provides is based on the difference between the service time of a cache hit and miss; the higher the cache hit ratio and the bigger the service time difference between a cache hit and miss, the bigger the performance improvement the cache provides. [48]

3.1.2 Advantages

There are three key advantages in caching: it makes web pages load faster, it reduces wide area bandwidth usage and it reduces the load placed on the origin servers. [48] From the end user's point of view, the biggest advantage of caching is the increase in the web application performance, meaning faster loading web pages. Distribution of dynamic content from an origin server usually requires more or less data computation based on certain criteria, until the resulting response can be send to the requesting client. If there are

a lot of similar clients requesting the same content, it would be useful to cache the result of the computation instead of doing it all over again with every request. This way, the processing power can be used to something else, which can greatly improve the maximum throughput of the application. [48]

From the application developer's point of view, the biggest advantage of caching is the reduced load on the origin servers. As explained before, program code written with the help of a framework is not necessarily the most optimized code possible, which means it lacks in performance. When the results of the important computations are cached, it does not really matter whether computing them takes 5 or 50 milliseconds, which makes the performance difference almost meaningless and permits the usage of the selected framework. [48]

Properly done caching offers tremendous advantages to an application and to the company. It saves time, money, bandwidth and processing power while increasing performance and stability of the application at the same time. None of the world's most popular modern web sites would work with the efficiency and stability they currently do if they did not utilize caching. Some could even say that caching is critical for making the whole Web usable at all. [48]

3.1.3 Disadvantages

Although caching is one of the most important building blocks of the modern Internet, it has also its downsides. First of all, cached information is always older than information retrieved or computed upon request. This can be controlled to some extent for example by defining **time to live (TTL)** parameters for cached items, but every once in a while users can still receive information that is out of date for one reason or another. It depends on the application how harmful this kind of situation is. In real-time applications even small delays in content freshness can seriously harm the intended functionality, which equals to bad user experience.

Secondly, many web application providers want to know exactly how their sites are used: which pages are most and least popular, who visits on what pages and when does this happen. Depending on the analyzing tools, some of these events may be hard to track in case the requested content is cached and the original request does not necessarily even reach the origin servers at all. [48]

Third, depending on how customized content the web application contains, caching can be utilized at different rates. A page customized specifically for Mary cannot be cached and returned to Jim, especially if it contains any personal data. In general, the more customized content an application contains, the less caching can be utilized. [48]

Finally, caching can also lead to problems with the legislation, since the cache can contain sensitive data, which should not be saved anywhere but on the origin servers. This kind of situation may limit the utilization of caching or even disallow it totally. [48]

3.1.4 Current utilization

At Conmio, caching is already utilized on many levels, thanks to the Conmio Cache plugin. Most of the Conmio web applications contain **domain objects** like *sections*, *articles* and *images*. These objects are usually created by parsing and interpreting the contents of one or more **Extensible Markup Language (XML)** or **JavaScript Object Notation (JSON)** feeds, that are provided by the customer. These processes are relatively expensive to the applications, and therefore every reasonably designed web application caches the results of these computations, that are the domain objects. This caching principle alone gives tremendous performance boost to the applications, because customer feeds can sometimes be slow to respond and also the amount of data to be processed may be large.

Conmio Cache has also a nice feature of being able to **asynchronously** update the contents of a cache after it has been once populated but the objects have expired. In these cases, the requesting client will receive the values from the cache immediately, after which the application updates the expired objects asynchronously and the next request will receive the updated objects. This makes it possible to always have immediately responding web application, as the expensive computations are mostly done behind the scenes. The downside in this is the occasional out of date response returned to the client because of sparse usage of the application.

In addition to caching of domain objects, other things can be cached too. Conmio Images plugin works together with the Cache plugin and caches the results of image manipulation processes. Conmio Cache has also the ability to cache fragments of HTML or other markup language, which further diminishes the need for repeated processing of the same content.

3.2 Advanced caching techniques

Following the general best practices of web application performance, it is clear that caching in its many forms is the option that has the most to offer for the Conmio web applications. However, it is important to understand all the possible ways of utilizing caching, recognize the trade-offs in each of these solutions and pick the most effective ones to be harnessed in a real application.

3.2.1 Improvement options

After discussing with the senior systems personnel of the company, at least 3 different potential caching improvement options were found: caching of general HTTP responses, caching of user group specific HTTP responses and caching of HTTP response fragments with Edge Side Includes markup language.

Out of these three, the caching of general HTTP responses is by far the most effective solution, since practically all requests would then lead to cache hits, unless the cache contents had expired or were not populated in the first place. However, this solution would require the same content to be returned to all clients, which would mean either non-customized content or heavy parsing of the returned content client-side, as in single-page applications.

The second way is to cache HTTP responses based on user groups or individual users. In this case, the content returned to different users could be customized to a degree. However, the more customized the content would be the less cache hits would occur, which would decrease the cache efficiency.

The utilization of Edge Side Includes markup language is the most fine-grained caching solution out of the three. With Edge Side Includes, individual page content fragments can be cached for separately specified lengths of time and the final HTTP responses are composed at **edge-level**, in the current case meaning usually an HTTP accelerator called Varnish. The obvious downside in this is the requirement for the HTTP accelerator software and the additional composing logic, which may be duplicated in the actual application.

3.2.2 Prioritization

Since the application in question is returning more or less customized content to the requesting clients, general HTTP responses cannot be cached without moving a lot of the parsing logic client-side, which would require a lot of work and fundamental changes to the application architecture. The only general responses that can be cached are the few AJAX requests these applications make, and they have already been cached with Varnish. On the other hand, in addition to being a fairly simple markup language, Edge Side Includes can be utilized alongside the other caching solutions, which makes it the most promising solution out of the three. In this case, the caching of user group specific HTTP responses seems to be somewhere in the middle in terms of usability and effectiveness, making it the second most promising solution.

3.2.3 Future possibilities

In the future web applications of the company, that have not yet any existing architecture design, the general HTTP response caching method can be utilized more effectively. This can be achieved by returning the majority of the content in JSON format and making the application single-page. It should be noted though, that the development process of single-page applications is much slower than with the traditional approach. However, utilization of a suitable framework may even out the situation.

Furthermore, multiple cloud computing services, like Google App Engine [49], are already offering automatic scaling of the resources for the applications deployed on their

platform. This way, the responsibilities concerning the server configurations and application scalability would be externalized outside the company, in theory removing the need of monitoring for high traffic peaks altogether. In practice though, this would also remove the control over monitoring and logging, therefore complicating multiple things, like tracking down bugs. Furthermore, it is unlikely that the automatic resource scaling would react fast enough in case of an unexpected high traffic peak, therefore diminishing the usability of this approach. Because of these reasons, utilizing cloud computing at this scale is inconvenient for the time being and cannot be considered as a real alternative in the current situation.

3.3 Varnish

Varnish is a web application accelerator also known as HTTP accelerator or caching HTTP reverse proxy. It can be installed in front of any server that communicates through HTTP and configured to cache the contents. Regardless of the architecture, retrieving the content from Varnish cache is usually at least 300 times faster than from the application. [50]

3.3.1 Functionality

Being a caching HTTP reverse proxy means that the requests from clients go through the Varnish before reaching the origin server, if reaching it at all. Varnish tries to answer to the requesting client from the cache first, and only if that cannot be done, is the request forwarded to the origin server, whose response is then cached and delivered to the client. The following requests to the same resource can be answered from the cache, provided that the content was cacheable and the cached version has not expired. An example of this scenario is illustrated in the Figure 3.1 [50]

By default, Varnish uses the *Cache-Control* header received alongside the origin server response to decide whether the content can be cached or not. There are a few conditions where Varnish will not cache, the most common one being a response containing cookies, which indicates user specific response. The default caching behaviour can be modified using policies written in **Varnish Configuration Language (VCL)**. It should be noted that Varnish is a software that does not require its own physical server to be utilized efficiently. [50]

3.3.2 Advantages

The single most obvious advantage of utilizing Varnish in front of a web application server is the tremendous performance improvement it is able to offer. All requests that can be answered from the cache will be returned to the clients faster and without the need of any processing from the application. This way, Varnish improves the perceived performance

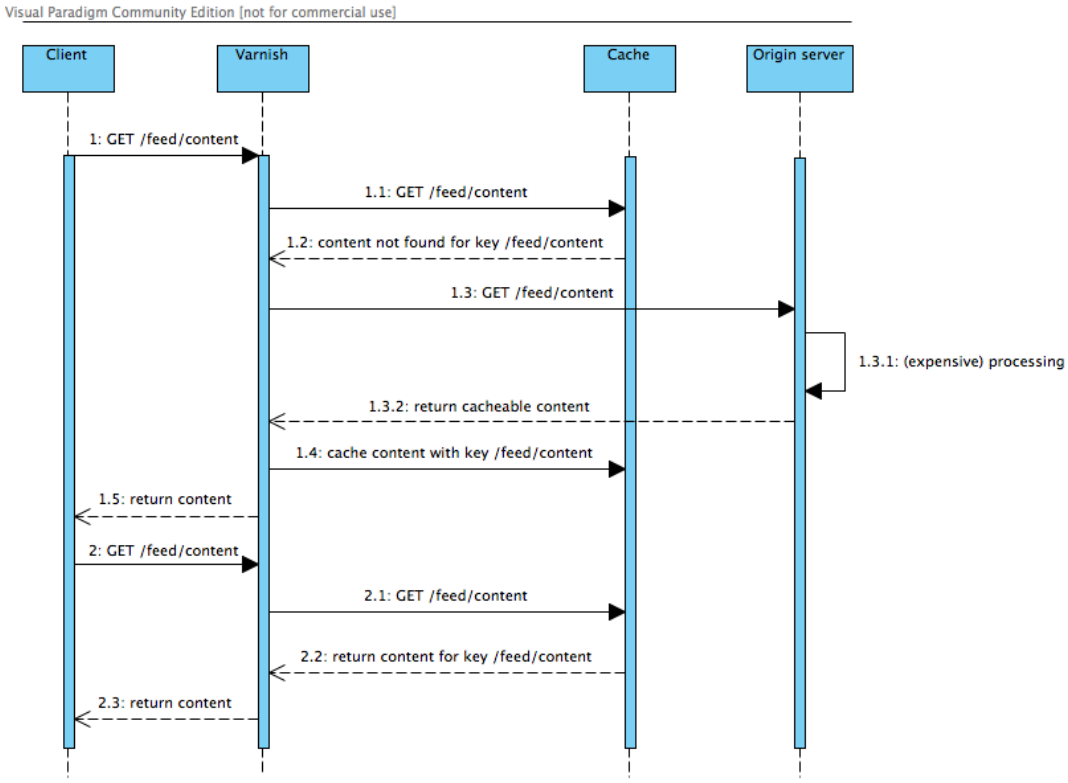

Figure 3.1: Varnish functionality.

experienced client-side while allowing the origin servers to use their resources elsewhere, thus increasing the maximum throughput of the application. [50]

Varnish also allows the configuration of caching behaviour through the Varnish Configuration Language, which makes it possible to create customized caching settings for each application, if necessary. This makes it possible to make caching more efficient, for example by grouping certain end user devices and allowing them to be cached with the same key. [50]

3.3.3 Disadvantages

The utilization of Varnish requires additional software and configuration, and in some cases also additional hardware. All this adds especially configuration and testing related work to the project, since the configuration settings have to be thoroughly tested in order to prevent caching errors between different clients. Especially the cache key generation functionality is very important, as it should take all relevant factors into consideration in terms of varying content.

On a practical level, many web application developers do not necessarily have a deep knowledge about Varnish or Varnish Configuration Language. Similarly, the systems personnel having knowledge about VCL do not necessarily have thorough understanding about the application. Without good communication, this may lead into a problematic

situation, that can recur every time a new application is about to be launched. In addition to this, from the maintenance point of view, it would be best to have certain default VCL settings used in all applications instead of customizing them separately each time, which reduces the possibilities for customized caching logic.

3.4 Edge Side Includes

Edge Side Includes (ESI) is an XML-based markup language designed for assembling resources in HTTP clients. Unlike some other in-markup languages, Edge Side Includes is designed to leverage tools like caches in order to improve the perceived performance while reducing the processing overhead on the origin server at the same time. It does this by allowing dynamic content assembly at the edge of the network, which can mean for example the client's browser, Content Delivery Network or an HTTP accelerator right next to the origin server. [51]

3.4.1 Functionality

The current and original Edge Side Includes markup language specification version 1.0 was introduced in 2001. It contains four major features: inclusion, variable support, conditional processing and exception and error handling. Out of these four, the most important feature in the scope of this thesis by far is the inclusion, which is described accurately while the other three are left with minimal attention. [51]

With the inclusion feature, the **ESI processor** can compose the final HTTP responses by assembling the included content, which is fetched from the network with separate requests. An include element should contain the full source URL with all the required query parameters for making a request that will return the fragment of HTML or other markup language, that should be used to replace the corresponding inclusion element in the final response. [51] Figure 3.2 contains a couple of examples of inclusion elements.

```
<esi:include src="http://example.com/1.html" alt="http://bak.example.com/2.html" onerror="continue"/>

<esi:include src="http://example.com/ search?query=$(QUERY_STRING{query})"/>
```

Figure 3.2: ESI inclusion element examples. [51]

In a nutshell, Edge Side Includes works like the following: When an HTTP request goes through ESI capable middleware to the origin server, additional headers are added to the request in order to make it possible for the origin server to recognize the ESI capability. In this case, the origin server will replace the actual content fragments with ESI inclusion elements in places where ESI functionality is to be utilized. The **response skeleton** then returns from the origin server to the ESI processor. At this point, the ESI processor assembles the final response by either including the actual HTML content fragments

from the cache or by making additional requests to the origin server through the source URLs defined in inclusion elements and then caching the results for future requests. The functionality of the Edge Side Includes is illustrated in the Figure A.1.

The other three features of ESI, variable support, conditional processing and exception and error handling, are features that can be used to enhance the inclusion feature. For example, additional logic can be written for the ESI processor in order to customize content or provide reasonable alternative responses in case one or more of the origin responses fail. However, the aim is to keep the ESI processor logic as simple as possible, which means the absence of any optional features. [51]

3.4.2 Advantages

The biggest advantage of ESI utilization is the ability to considerably reduce the workload on the origin servers without moving it client-side. The more ESI inclusions are utilized on a web page, the less work the origin servers have, which frees up resources for serving more clients.

Compared to some other caching techniques, ESI inclusion fragments have their own metadata, which can contain information about things like cache expiration time and alternative source URL. This makes it possible to adjust the caching times of different content fragments based on their updating frequency. For example, a markup fragment containing the menu of the application updates rarely, thus allowing caching time of multiple hours, whereas social discussion messages may be cached only for five seconds or so. This makes the caching almost invisible to the client, as the appropriate content seems to be updating in real-time despite the caching. [51]

Although the fine-grained markup fragment caching described here is possible also in the application with the help of Conmio Cache plugin, the difference here is that the ESI cache is faster and it does not put any load on the application. Furthermore, if the ESI processor is running on a different physical server, it does not put any load on the whole origin server.

3.4.3 Disadvantages

First of all, utilization of ESI requires additional software that understands the ESI language, namely ESI processor. In addition, the software has to be configured in order to work properly alongside other existing server software, like Apache HTTP server and Apache Tomcat.

Secondly, if additional features besides inclusion are utilized, the logic part of ESI processor may become complex and duplicated with the logic inside the application. This makes it hard to maintain and reuse in other projects.

Third, it should be precisely defined which features of a client are affecting to the content of an ESI fragment, so that all those features are included in the cache key, preventing caching of varying content with the same key but still keeping the cache hit ratio as high as possible. This obviously adds some development and testing work to the project.

3.4.4 Edge Side Includes and Varnish

Varnish can be used to cache many kind of content ranging from full HTTP responses to small content fragments. To help with fragment caching, Varnish implements a small subset of ESI features, namely inclusion and removal elements. Therefore, Varnish can do ESI processing, provided that only the supported elements are used in the markup. [50] This is enough for the current purpose.

3.4.5 Edge Side Includes and Conmio Modules

Conmio Modules plugin was originally designed to encapsulate the logic and markup of a content markup fragment. These fragments can present elements like headers, footers, menus and news listings. The goal was to allow easy adding, moving and removing of individual content elements on a web page.

When considered alongside Edge Side Includes, Conmio Modules seems to map well with the ideology: both techniques are designed to produce a specific markup fragment in one way or another. Therefore, support for ESI was added to the Conmio Modules plugin by implementing a sniffer that checks whether the current request has come through an ESI capable middleware. If yes, the module will not execute its normal functionality but instead produce a suitable ESI inclusion tag, which can be used to fetch the actual content of the module. This way, the processing of the module content has to be done only when the ESI cache does not have fresh enough version of the requested content fragment.

To the developer, the switch to ESI modules is extremely simple: instead of calling the module *module*, it can be called *esiModule* instead, which activates the ESI functionality of the plugin. In addition to this, ESI modules work like regular modules in cases where ESI processor is not present in the application pipeline, thus allowing the full functionality of the modules either way.

3.5 User group specific HTTP response caching

In short, user group specific HTTP response caching means caching of HTTP responses, whether they contain HTML, XML, JSON or whatever other markup language. Although this could be done in Varnish, there are some big advantages when doing it in the origin server instead.

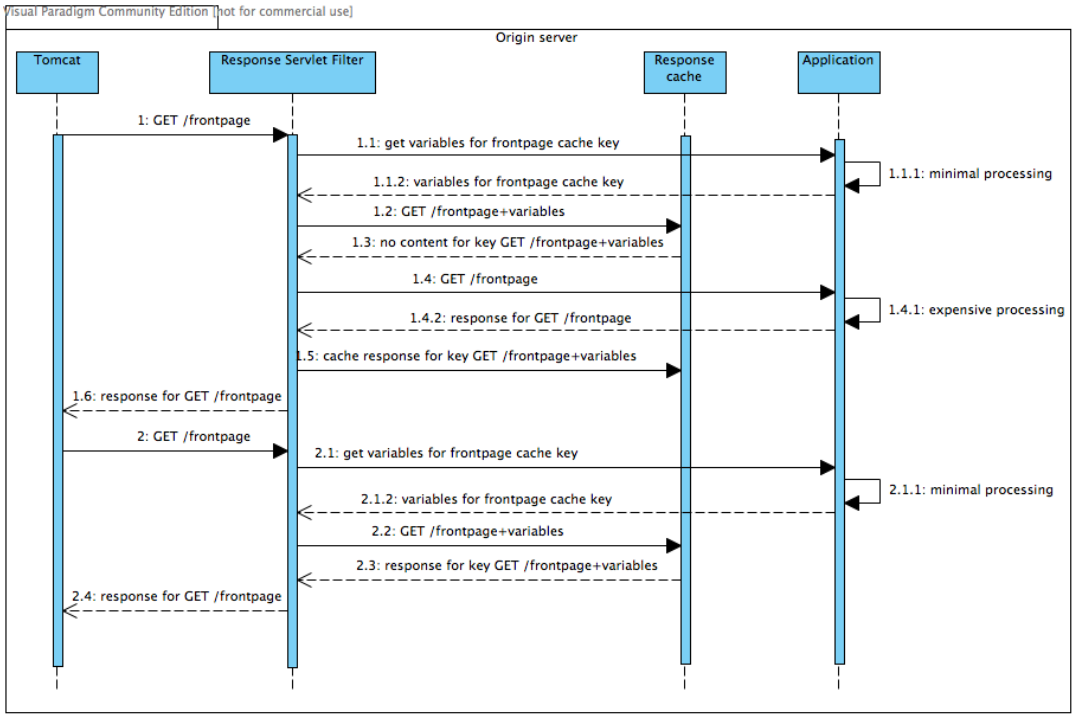

Figure 3.3: A servlet filter response cache functionality.

3.5.1 Functionality

In practice, caching of user group specific HTTP responses in a Java-based web application is convenient to do with a Java **Servlet filter** implementation. A Servlet filter is able to dynamically intercept requests and responses and transform or use the information contained in them. In this case, the filter can be used to decide whether a certain request can be answered from the cache instead of rendering the response in the actual application. A sequence diagram of the response cache functionality is illustrated in the Figure 3.3. [52]

3.5.2 Advantages

The obvious advantage of caching user group specific HTTP responses in the first place is the improvement in perceived performance while decreasing the processing workload on the server. Compared to doing the response caching in Varnish, doing it in a Servlet filter allows access to functions of the application, which can therefore be utilized when generating the cache key, thus determining the user groups for each cache entry. When done like this, the key generation logic can be written in Java or Groovy by the application developer without deep knowledge about the filter functionality or Varnish Configuration Language. This also makes it easier to keep the majority of the caching logic as close to the application as possible and allows the utilization of default VCL settings in Varnish.

3.5.3 Disadvantages

This technique has the usual downsides of caching in general: the content is not as fresh as it could be, additional logic has to be written for the cache key generation and the whole caching functionality has to be thoroughly tested with multiple end devices. However, the downsides in this technique seem to be scarce compared to some of the other caching techniques.

4. EDGE SIDE INCLUDES UTILIZATION

4.1 Optimization steps

After the addition of ESI support into the Conmio Modules plugin, it is clearly the most practical way to utilize ESI in Conmio web applications. In practice, utilization of ESI requires three steps: extraction of suitable content fragments into modules using Conmio Modules plugin, extraction of all necessary information regarding the variation of the module content into explicit variables and configuration of Varnish HTTP accelerator in front of the application origin servers.

The first two steps are left to the application developers, while the third step is mainly directed towards the systems personnel. However, the VCL settings for Varnish should be made in co-operation between the systems personnel and the application developers in order to ensure the proper functionality of the application. This is essential especially if achieving the proper functionality requires customized logic for Varnish. However, a default VCL configuration settings file was created by the systems personnel to be utilized in all projects whenever possible, thus reducing the need for customization between different projects. These default settings are utilized also in the current application.

4.2 Performance before optimization

```
Concurrency Level:      50
Time taken for tests:   170.011 seconds
Complete requests:      10000
Failed requests:        0
Write errors:           0
Total transferred:      421360000 bytes
HTML transferred:       418740000 bytes
Requests per second:    58.82 [#/sec] (mean)
Time per request:       850.056 [ms] (mean)
Time per request:       17.001 [ms] (mean, across all concurrent requests)
Transfer rate:          2420.34 [Kbytes/sec] received
```

Figure 4.1: The maximum throughput of the web application front page through Apache HTTP server before optimization.

The first thing to do is to measure the starting level of the web application performance before any optimization has taken place. In order to achieve the most reliable as well as

practical results, the front page of a real life web application is used in all performance measurements that follow. Luckily, the same optimized application can be used to measure all the different levels of optimization, as the ESI functionality of Conmio Modules can be switched off, causing them to work like regular modules, which is performance-wise very closely the same as if Conmio Modules were not utilized at all.

The maximum throughput of the non-optimized web application front page is measured through Apache HTTP server using ApacheBench and dedicated server as described in the chapter 2. Special attention is paid to the *Requests per second* value. The benchmark results of the fifth test run are visible in the Figure 4.1.

According to the general Conmio web application performance guidelines, the value of 80 requests per second per page is considered to be good enough for most applications. The testing machine seems to be a bit less powerful than the actual server machines, since it is able to serve approximately 59 requests per second compared to the value of roughly 80 requests per second of the actual server machines. However, the maximum throughput improvement factor can be untangled with the test machine despite the inferior performance.

It should be noted that Varnish was actually already utilized in the application at this point for caching the general AJAX responses. Therefore, the front page should have been fetched through Varnish rather than through Apache HTTP server. However, the maximum throughput value for the front page seemed to be practically the same whether the Varnish was utilized or not, thus making the difference in measurement configuration meaningless at this point.

4.3 Edge Side Includes optimization

A wireframe illustration of the web application front page structure before splitting it into modules can be seen in the left side of the Figure B.1. By inspecting the front page structure more thoroughly, it becomes clear that the structure can be split into separate modules in multiple ways. On the first try, 10 visible logical modules can be found: splash screen, top ad, header with top menu, poll, external, vote, news listing, social discussion, bottom ad and footer. In addition to these, there are a few additional modules containing required but non-visible information, like analytics and meta-information. For clarity's sake, these non-visible modules and splash screen module are left out of the illustrations.

When delving deeper into the logic of different modules, and remembering the fact that the more content is inside ESI modules, the less work is left for the application, it seems reasonable to merge a couple of modules together. After the merging, basically all front page content is inside ESI modules. The front page now consists of 8 visible logical modules: top ad, header with top menu, poll, external, vote, news listing with social discussion, bottom ad and footer. The final separation of the visible content modules can be seen in the middle of the Figure B.1.

The total count of the front page modules at this point is 13, which means there are 5 modules not visible in the illustrations. For now, it is sufficient to say that these non-visible modules do not contain anything that would have a major effect on the application performance. Nevertheless, it is important to acknowledge the total number of the modules on the front page.

4.4 Performance after Edge Side Includes optimization

```
Concurrency Level:      50
Time taken for tests:   64.554 seconds
Complete requests:      10000
Failed requests:        0
Write errors:           0
Total transferred:      423797101 bytes
HTML transferred:       420926240 bytes
Requests per second:    154.91 [#/sec] (mean)
Time per request:       322.771 [ms] (mean)
Time per request:       6.455 [ms] (mean, across all concurrent requests)
Transfer rate:          6411.11 [Kbytes/sec] received
```

Figure 4.2: The maximum throughput of the ESI optimized web application front page through Varnish.

When comparing the origin server responses for the application front page depending on whether the complete response or just the response skeleton is being rendered, a tremendous difference in the amount of data can be identified; the complete response size is 8661 bytes while the response skeleton size is only 807 bytes. In other words, the response skeleton is less than 1/10th of the complete response size.

It is safe to assume that rendering the response skeleton puts much less load on the application than rendering the complete response, since almost all actual content is inside ESI modules. However, this is not the whole truth since the contents of the ESI modules have to be also rendered every time their cached versions expire. How often this happens depends on the module. The *time to live* values used for different modules can be seen in the right side of the Figure B.1.

It seems there are three modules containing dynamic content: poll, vote and news listing with discussion. The TTL values of these modules are 5, 15 and 5 seconds, so they are rendered fairly often compared to the other modules of the front page. However, from the application performance perspective, even modules with TTL value of 5 seconds need rendering quite rarely, which makes the rendering processes of these modules almost meaningless in the scope of this thesis.

The maximum throughput of the ESI optimized web application front page is measured through Varnish with previous module TTL values. The benchmark results of the fifth test run can be seen in the Figure 4.2.

A clear improvement in the *Requests per second* value can be seen, as the new value is roughly 3 times the original. Additionally, the maximum throughput of the front page response skeleton is measured through Apache HTTP server and the results are illustrated in the Figure 4.3.

```
Concurrency Level:      50
Time taken for tests:   51.081 seconds
Complete requests:      10000
Failed requests:        0
Write errors:           0
Total transferred:      35110000 bytes
HTML transferred:       30810000 bytes
Requests per second:    195.77 [#/sec] (mean)
Time per request:       255.406 [ms] (mean)
Time per request:       5.108 [ms] (mean, across all concurrent requests)
Transfer rate:          671.23 [Kbytes/sec] received
```

Figure 4.3: The maximum throughput of the web application front page skeleton through Apache HTTP server.

It seems that despite almost all front page content is now inside ESI modules and the response skeleton is very simple, the rendering process of the skeleton is still fairly heavy. This is probably due to Grails' core features, like utilization of Sitemesh layouts.

```
Concurrency Level:      50
Time taken for tests:   10.197 seconds
Complete requests:      10000
Failed requests:        0
Write errors:           0
Total transferred:      3573570 bytes
HTML transferred:       1641640 bytes
Requests per second:    980.65 [#/sec] (mean)
Time per request:       50.987 [ms] (mean)
Time per request:       1.020 [ms] (mean, across all concurrent requests)
Transfer rate:          342.23 [Kbytes/sec] received
```

Figure 4.4: The maximum throughput of the web application updated poll results JSON through Apache HTTP server.

In order to illustrate the power of caching general JSON responses in Varnish compared to caching of HTML fragments with ESI, the maximum throughput of updated poll results JSON responses is measured first through Apache HTTP server and then through Varnish in a similar manner than before. The results for Apache HTTP server are visible in the Figure 4.4 while the results for Varnish can be seen in the Figure 4.5.

Even the *Requests per second* value measured through Apache HTTP server is very high, but the addition of Varnish pushes the value through the roof. Over 6000 requests

```
Concurrency Level:      50
Time taken for tests:   1.661 seconds
Complete requests:      10000
Failed requests:        0
Write errors:           0
Total transferred:      4390539 bytes
HTML transferred:       1647708 bytes
Requests per second:    6019.01 [#/sec] (mean)
Time per request:       8.307 [ms] (mean)
Time per request:       0.166 [ms] (mean, across all concurrent requests)
Transfer rate:          2580.73 [Kbytes/sec] received
```

Figure 4.5: The maximum throughput of the web application updated poll results JSON through Varnish.

served in a second is a huge value for the test machine, and should be sufficient for almost anything.

It seems clear that rendering and caching of general JSON responses is the most efficient way to provide responses from the application. It should be noted though, that the response under inspection is fairly simple piece of JSON, and the performance probably decreases a bit when rendering more complex responses.

4.5 Summary

Using ESI optimized version of the application is clearly beneficial to the application performance. However, it seems that in the current solution the front page skeleton rendering performance is preventing the ESI solution from reaching its full potential. This is due to the fact that the front page skeleton is rendered on the origin server with every request, even though the actual content fragments are cached in Varnish.

A way to improve the situation would be to cache also the page skeleton in addition to the content fragments. This could be done in Varnish, but it would require customized VCL configurations, which would probably lead to duplicated logic between Varnish and the application itself. As these are things to be avoided, another solutions are explored.

The rendering process of general JSON responses seems to be so fast even without any additional caching, that it would not easily become the performance bottleneck in this case. The addition of Varnish in front improves the performance even further, making it practically impossible to become the bottleneck in any case.

5. RESPONSE CACHING UTILIZATION

5.1 Background

In addition to Varnish, the response skeleton can be cached also elsewhere. One viable option in this case is the origin server, which can utilize a Servlet filter implementation wrapped around the application in order to determine whether a certain request can be answered from the cache instead of passing the request to the actual application.

5.2 Optimization steps

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.5"
         metadata-complete="true"
         xmlns="http://java.sun.com/xml/ns/javaee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd">

    <display-name>/@grails.project.key@</display-name>

    ...

    <filter>
        <filter-name>responseCachingFilter</filter-name>
        <filter-class>com.conmio.ResponseCachingFilter</filter-class>
        <init-param>
            <param-name>cacheName</param-name>
            <param-value>responseCache</param-value>
        </init-param>
    </filter>

    <filter-mapping>
        <filter-name>responseCachingFilter</filter-name>
        <url-pattern>/*</url-pattern>
        <dispatcher>REQUEST</dispatcher>
        <dispatcher>FORWARD</dispatcher>
    </filter-mapping>

    ...

</web-app>
```

Figure 5.1: Required web.xml configuration settings for the response cache.

Like with the ESI modules, the utilization of response caching requires three steps: extraction of all necessary information regarding the variation of the page content into explicit variables, creation of a Servlet filter implementation with suitable key calculation functionality and configuration of a cache for the responses.

The first step is easy after the utilization of ESI modules, since all relevant variables have already been extracted and are available for calculating the cache key. The second step is also pretty easy, since there already is at least one almost suitable Servlet filter implementation, called SimplePageCachingFilter [53], for the **Ehcache** [54]. Only the

functions for calculating the cache key and selecting which URLs to cache need to be overridden in order to achieve the desired functionality. The required code for these overrides can be seen in the Figure C.1.

The third step is closely related to the second step, as the cache for SimplePageCachingFilter can easily be configured with *web.xml* and *ehcache.xml* configuration files placed inside the Grails project. The relevant code for these files can be seen in the Figures 5.1 and 5.2.

```
<ehcache>

    <diskStore path="java.io.tmpdir"/>

    <defaultCache
            maxElementsInMemory="10"
            eternal="false"
            timeToIdleSeconds="5"
            timeToLiveSeconds="10"
            overflowToDisk="true"
            />

    <cache name="responseCache"
           maxElementsInMemory="1000"
           eternal="false"
           timeToIdleSeconds="2"
           timeToLiveSeconds="2"
           overflowToDisk="false">
    </cache>

</ehcache>
```

Figure 5.2: Required ehcache.xml configuration settings for the response cache.

As can be seen from the *ehcache.xml*, the response cache *time to live* value is only 2 seconds and the *eternal* configuration option is set to *false*, which means a response received from the cache can never be more than 2 seconds old. During the low traffic window, the cache is pretty meaningless because it is mostly empty and cache hits are rare. However, when the high traffic peak arrives, cache hits become more frequent and the application has to render a certain kind of page skeleton only once in every 2 seconds instead of doing it 700 times per second. This greatly reduces the need for repetitive processing on the origin servers, and thus improves the application scalability.

5.3 Performance after response cache optimization

After configuring the ResponseCachingFilter to filter requests directed towards the front page of the application along with the actual cache settings, it is safe to assume a tremendous performance improvement is possible due to the major decrease in the front page response skeleton processing frequency during a high traffic peak.

The maximum throughput of the response cache optimized web application front page response skeleton with previously defined cache settings is measured through Apache

```
Concurrency Level:      50
Time taken for tests:   7.133 seconds
Complete requests:      10000
Failed requests:        0
Write errors:           0
Total transferred:      34877530 bytes
HTML transferred:       30825405 bytes
Requests per second:    1401.95 [#/sec] (mean)
Time per request:       35.665 [ms] (mean)
Time per request:       0.713 [ms] (mean, across all concurrent requests)
Transfer rate:          4775.06 [Kbytes/sec] received
```

Figure 5.3: The maximum throughput of the response cache optimized web application front page response skeleton through Apache HTTP server.

HTTP server. The benchmark results of the fifth test run can be seen in the Figure 5.3.

As assumed, the front page response skeleton rendering performance seems to have improved tremendously, as the *Requests per second* value is now roughly 9 times higher than previously. With this improvement in place, the performance bottleneck have been moved elsewhere, as the skeleton rendering performance is now sufficient for the required performance level.

Next, the maximum throughput of the ESI and response cache optimized web application front page is measured through Varnish. The benchmark results of the fifth test run are illustrated in the Figure 5.4

```
Concurrency Level:      50
Time taken for tests:   17.245 seconds
Complete requests:      10000
Failed requests:        0
Write errors:           0
Total transferred:      423966991 bytes
HTML transferred:       421343360 bytes
Requests per second:    579.88 [#/sec] (mean)
Time per request:       86.225 [ms] (mean)
Time per request:       1.724 [ms] (mean, across all concurrent requests)
Transfer rate:          24008.78 [Kbytes/sec] received
```

Figure 5.4: The maximum throughput of the ESI and response cache optimized web application front page through Varnish.

The *Requests per second* value is now almost 10 times higher than in the beginning. However, at the same time, the value is much lower than that of rendering just the response skeleton. It might be that the aggregation of the final response in Varnish actually requires so much processing power that it becomes the performance bottleneck in this case.

In order to see what happens to the performance if the ESI modules are not utilized but the front page is still fetched through Varnish from the response cache optimized origin

server, the performance is measured with ApacheBench as before. The benchmark results of the fifth test run for this configuration are illustrated in the Figure 5.5.

```
Concurrency Level:      50
Time taken for tests:   17.398 seconds
Complete requests:      10000
Failed requests:        0
Write errors:           0
Total transferred:      422419148 bytes
HTML transferred:       419341862 bytes
Requests per second:    574.78 [#/sec] (mean)
Time per request:       86.990 [ms] (mean)
Time per request:       1.740 [ms] (mean, across all concurrent requests)
Transfer rate:          23710.75 [Kbytes/sec] received
```

Figure 5.5: The maximum throughput of the response cache optimized web application front page through Varnish.

It seems that the perceived rendering performance through Varnish is approximately the same whether the ESI modules are utilized or not. However, the utilization of ResponseCachingFilter seems to have a big effect on the application performance in both cases. In fact, it is so big that it can make the utilization of Varnish unnecessary. Therefore, one more test set is run by requesting the complete front page through Apache HTTP server from the response cache optimized web application. The benchmark results of the fifth test run with this configuration can been seen in the Figure 5.6.

```
Concurrency Level:      50
Time taken for tests:   14.264 seconds
Complete requests:      10000
Failed requests:        0
Write errors:           0
Total transferred:      421444980 bytes
HTML transferred:       419073146 bytes
Requests per second:    701.07 [#/sec] (mean)
Time per request:       71.320 [ms] (mean)
Time per request:       1.426 [ms] (mean, across all concurrent requests)
Transfer rate:          28853.66 [Kbytes/sec] received
```

Figure 5.6: The maximum throughput of the response cache optimized web application front page through Apache HTTP server.

The performance has improved a bit compared to the previous configurations, but the difference is not radical. However, it seems that Varnish is actually just slowing things down in this case, due to the CPU processing power required for composing the final response.

As a comparison, the maximum throughput of the updated poll results JSON responses from the response cache optimized application is measured through Apache HTTP server. The benchmark results of the fifth test run are visible in the Figure 5.7.

```
Concurrency Level:      50
Time taken for tests:   4.690 seconds
Complete requests:      10000
Failed requests:        0
Write errors:           0
Total transferred:      3603960 bytes
HTML transferred:       1671837 bytes
Requests per second:    2132.27 [#/sec] (mean)
Time per request:       23.449 [ms] (mean)
Time per request:       0.469 [ms] (mean, across all concurrent requests)
Transfer rate:          750.45 [Kbytes/sec] received
```

Figure 5.7: The maximum throughput of the response cache optimized web application updated poll results JSON through Apache HTTP server.

It seems that the response cache optimization is not as efficient for the general HTTP response caching as Varnish, since the maximum throughput value of the response cache optimized version is only about 1/3rd of the Varnish optimized version. However, the value of over 2000 served requests per second can still be considered as a good value.

5.4 Performance in a realistic high traffic simulation

Most of the performance measurements so far have focused solely on the maximum throughput of the web application front page. This is appropriate, since the front page rendering process was the problematic performance bottleneck in the first place. However, now when the bottleneck has been removed, it would be useful to gain some knowledge about the actual real life performance improvement achieved during the process.

In order to find out how well the optimized application performs under realistic high traffic conditions compared to the original version, a more sophisticated load testing tool, JMeter, is utilized. In order to simulate the server traffic as realistically as possible, the following specifications are used: There are 50 concurrent desktop users and 100 concurrent mobile users on the application front page simultaneously. Each major smartphone operating system is represented, meaning iOS, Android and Windows Phone devices are accessing the application. Poll votes are sent for both available options with all end devices. In addition to these, there are 50 concurrent mobile users browsing through all of the major pages of the application, including news, discussion and poll pages. Generally speaking, there are the total of 200 concurrent users in the application doing the most common tasks in very high pace.

First, the original application is deployed on the same test machine as before. The JMeter is run from the student's computer, so that it will not cause any additional load to the test machine. The simulation is run until approximately 15000 requests have been made. The general information about the fifth test run is illustrated in the Figure D.1.

Probably the most relevant information here is the *Throughput* value, which is illus-

Figure 5.8: The maximum total throughput of the non-optimized web application through Varnish.

trated also in the Figure 5.8. It seems to be just shy of 8816 requests per minute, translating to the value of approximately 147 requests per second. This value represents the original maximum total throughput of the application, before any of the optimization steps explained in this thesis were executed.

The value seems to be higher than expected for the non-optimized application. The explanation is that the AJAX requests have already been optimized by caching them in Varnish, causing them to be very fast, thus raising the total throughput value of all requests. To further explain the situation, the average response times of different requests are illustrated in the Figure D.2.

As can be seen, the front page of the application seems to have the longest average response time, a bit over 3000 milliseconds, with all of the main section pages following not too far behind. The fastest response time is recorded for the AJAX request for fetching the updated poll results, with the value of just 68 milliseconds.

Next, the optimized application is deployed on the test machine and the same simulation is run again. It should be noted, that the response cache is now configured to cache all of the main section pages of the application in addition to the front page. The general information about the fifth test run is illustrated in the Figure D.3.

Figure 5.9: The maximum total throughput of the optimized web application through Varnish.

As can be seen from the Figure 5.9, the *Throughput* value seems to be now as high as 20677 requests per minute. This translates to the value of approximately 345 requests per second, which is about 2.3 times the original value. Furthermore, the average response times of different requests made to the optimized web application are illustrated in the Figure D.4.

The response times of different requests have flattened noticeably, as the front page response time is now only 512 milliseconds compared to the original value of over 3000 milliseconds. In other words, the average response time for the optimized application front page is only 1/6th of the original. Other main section pages are largely in par with the front page, although participants page response time is a bit higher with the value of 914 milliseconds. This is due to the fact that the participants page contains a lot of content that is not inside ESI modules because of certain technical reasons.

Figure 5.10: The maximum total throughput of the optimized web application through Apache HTTP server.

Finally, the same simulation is run once more, but this time for the response cache optimized application that caches also the general AJAX responses. In this version, Varnish can be totally dismissed, which simplifies the application architecture. The general information about the fifth test run is illustrated in the Figure D.5

The *Throughput* value seems to be even higher than before, as can be seen from the Figure 5.10. The value is approximately 21423 requests per minute, which translates to the value of approximately 357 requests per second. Compared to the previous configuration, the value is higher, though the difference is minor. Likewise, the average response times of different requests made to the response cache optimized application are illustrated in the Figure D.6.

It seems that the response times for most of the requests have decreased a bit. The unexpected thing is the fact that also the AJAX requests' response times have decreased although Varnish is no longer used for caching them. The explanation to this is unknown at this point, but it may be caused by things like JVM optimization procedures or the measuring tool itself.

5.5 Summary

It seems that out of the two advanced caching techniques under inspection, utilization of response caching seems to be the single more efficient option for increasing the maximum throughput of the application. This is due to the fact that the response cache seems to be more efficient especially with the user group specific HTTP responses, which are constituting most of the current application content. On the other hand, Varnish is more efficient with the general AJAX responses, but as there are not so many of them in the current application, the benefit of caching them is minor.

However, the efficiency of the ESI solution could be improved for example by externalizing the ESI processing from the Varnish to the CDN. This way, the processing power of the ESI processor would always be sufficient without reducing any power from the origin servers. The downside in this approach is that the CDN should support ESI processing, which is not the case with the current CDN provider of the company.

Another possibility for gaining a similar effect would be to move the Varnish on its own server machine, thus giving it more CPU-time without reducing it from the application. However, this would also increase the number of servers in use, which was a thing to be avoided.

Furthermore, the ESI modules themselves could be split differently and different *time to live* values could be used on those modules. One especially efficient way could be to create only one module per page, which would probably lead to better performance because of the simplicity of the content aggregation process. However, this would happen with the downside of reducing the usability of many Grails' core features, including Sitemesh layouts and resource rendering tags. In other words, what could be gained performance-wise would be lost in maintainability and developer happiness, which makes this an option to be avoided.

When reflecting back to the original goal of the thesis, improving the maximum throughput of the application as much as possible, it seems that ResponseCachingFilter alone can do most of the task. However, as the real life web application contains also other pages than front page, the ESI modules become more useful especially if multiple pages contain identical modules. Furthermore, if the response aggregation process would be externalized to its own server, the efficiency of the ESI solution would increase even more.

When comparing the maximum total throughput performance of the original application to the the optimized application in a real environment under realistic traffic, it seems that the optimized application performs around 2.3 times better regardless of the usage of Varnish. However, it is impossible to say how close to the truth this result is because the current performance bottleneck is unknown; it might be the network between the testing machines or even the JMeter measuring tool itself, for example. All in al, it is undeniable that a noticeable improvement in the maximum total throughput of the application was achieved regardless of the selected measurement tool or configuration.

6. CONCLUSIONS

6.1 Selected optimization techniques

The different optimization techniques presented in this thesis were based on the standard front end performance guidelines by Steve Souders. Out of all the possibilities, caching was seen as the most promising candidate for fulfilling the high performance requirements of the web application in question. Out of all the caching possibilities, three specific techniques of advanced caching were identified: caching of general HTTP responses, caching of user group specific HTTP responses and caching of HTTP response fragments by utilizing Edge Side Includes markup language.

6.1.1 Theoretical findings

By inspecting the theory behind each of these techniques, Edge Side Includes was selected to be the first technique to be utilized in the application, as it promised to offer just what was required: a way to reduce the load of the origin servers without moving the work client-side but rather doing some of it in the already existing middleware layer. Another factor supporting this solution was the fact that Varnish, the HTTP accelerator software capable of doing also ESI processing, was already utilized in front of the application for caching some of the general AJAX responses. Therefore, not much configuration was required for neither the hardware nor the software side of the application architecture in order to implement the support for Edge Side Includes.

The second most useful caching technique for the current application was identified to be the caching of the user group specific HTTP responses. With this technique, the logic for composing the customized responses can still reside on the origin servers, but the frequency for executing it has diminished tremendously. Instead, by calculating the cache key for each request in the application, which requires only the minimal amount of processing, many of the requests could be responded from the cache instead of processing the same response all over again in the application. Furthermore, this technique can be utilized alongside Edge Side Includes or on its own.

The third technique, caching of general HTTP responses, was immediately identified as the most efficient way of caching HTTP responses in general. This is due to the fact that practically every request would lead to a cache hit because the response would be the same for everyone. Unfortunately, in the current application the utilization possibilities

of this technique are limited, since the client would require complex logic for parsing the general responses. However, this technique could be utilized in the few AJAX requests made by the application. In fact, this was already done before the thesis project even began, and it is also the original reason for the existence of Varnish in the web application architecture.

6.1.2 The results of the empirical studies

As planned, the first optimization step was to start utilizing Edge Side Includes markup language in the application. This required the modularization of the page content using Conmio Modules plugin, extraction of all information affecting to the module content into explicit variables and creation of suitable Varnish Configuration Language settings for the Varnish in front.

After completing these steps and comparing the benchmark results between the original application and Edge Side Includes optimized version, it was calculated that the latter could serve approximately three times the number of requests for the application front page compared to the former. While this clearly was an improvement in the maximum throughput of the application, it was less than what was expected. The reason for the inefficiency was pinpointed later to be the rendering process of the response skeleton in the application.

Next, the user specific HTTP response caching mechanism was implemented by extending an existing Java Servlet filter implementation for response caching and overriding suitable methods from the original implementation. In addition to these, a cache for the responses was configured. After completing these steps, the maximum throughput of the front page response skeleton increased to almost tenfold number compared to the original version.

After removing the performance bottleneck from the skeleton rendering process, the maximum throughput of the application front page was measured again. The number of served requests was now around ten times the value of the original application. During this time, it was also noticed that the utilization of Edge Side Includes is actually pretty meaningless if the HTTP response caching filter is utilized in the application. This is due to the fact that in the current server configuration, both the Varnish and the application itself reside on the same machine. Therefore, the reduced amount of processing work in the application is actually just moved to the Varnish, which uses the same hardware for its own processing tasks it has to do before the final response can be sent to the client.

Finally, the maximum total throughput improvement of the application was measured by simulating a realistic high traffic peak situation for both the original and the optimized application. It turned out that the optimized application could serve approximately 2.3 times the number of requests compared the original application. Furthermore, the result was practically the same whether Varnish was utilized in front of the response cache

optimized application or not.

6.1.3 Measurement reliability

Most of the performance measurements of this thesis were made by utilizing only a simple command-line tool ApacheBench. In addition to this, the performance of the application was measured under realistic high traffic conditions with more advanced load testing software JMeter. Furthermore, the expected performance changes were estimated by measuring and comparing the sizes of different kind of responses, like complete response and response skeleton.

ApacheBench gives very close to realistic results only when the requested content is exactly the same for everyone. In the current application, it basically means only the few AJAX requests for updating the poll results and Twitter feed. All the other content varies more or less depending on the client, and the varying client is hard to simulate properly with ApacheBench.

Taken as an example, the front page of the application consists of four variables that affect to the content: three of them are boolean values and one is the poll answer value, which usually has two to five different answer options and no answer at all. Basically this means that there are 24 to 48 different versions of the front page that should be rendered; not just the one ApacheBench is requesting.

However, this also means that 48 is the absolute maximum number of times a single server has to render the front page during the defined cache *time to live* time, which was now 2 seconds. This translates to the maximum of 24 rendered front pages in a second. If this is compared to a high traffic peak situation of 700 requests per second, if they all are requesting the front page, the required effort from the application is now only 1/30th of the original. When taken into the account that this is the absolute worst-case scenario, as the other pages are simpler and have less variables, it is pretty safe to say that ApacheBench results are close enough to the truth for the current purpose.

Nonetheless, in order to measure the true effect of the previous optimization steps in a real web application, a more realistic high traffic simulation was created with JMeter and run for both the original and the optimized application. The result was that the optimized application performed around 2.3 times better than the original regardless of the usage of Varnish in front.

The reliability of this particular measurement setting is probably close to the truth, but as the current performance bottleneck is unknown, nothing can be said for sure. For example, the internal network of the office could have slowed the response times down even though both computers were connected to the same network with Ethernet cables. Furthermore, the JMeter software itself seemed to slow down some time after making 15000 requests to the application, which can also affect to the measurement results. In addition, JVM is capable of doing its own optimization procedures according to logic

unknown, thus reducing the comparability of different performance results a bit. Because of these reasons, the maximum total throughput improvement factor presented here can only be considered as an approximation of the real value.

6.1.4 Further improvement possibilities

It is undeniable that a noticeable improvement in the maximum throughput of the application was achieved during the thesis project regardless of the measurement configuration. However, there are a few ways the performance could be improved even further, though they all contain some trade-offs. Generally, more advanced traffic simulations and measuring tools could be utilized in search for the optimal balance between the following improvement options.

First of all, in order to grant the maximum processing power for the ESI processor, the content aggregation process could be externalized from the Varnish to the CDN. This way, the processing power of the ESI processor would always be sufficient without reducing any power from the origin servers. The downside in this approach is that the CDN should support ESI processing, which is not the case with the current CDN provider of the company.

A similar effect could be achieved also by moving the Varnish to its own server machine from the origin server, thus allowing it more CPU-time without reducing it from the application. However, the total number of the required servers and the response times between Varnish and Apache HTTP server would probably increase in this case.

Third, the split into ESI modules could be done differently, for example by creating only one module per page, thus reducing the response aggregation workload in Varnish. However, this would also diminish the usability of many Grails' core features, leading to messier code that is harder to maintain.

Fourth, if the application was turned into a single-page application, the servers could utilize general HTTP response caching. This way, the performance of the servers would be pretty much as close to the optimal as possible. However, this would require heavy rewriting of the current application code, though it could be utilized more easily in future applications.

Multiple small adjustments to the application could make an impact to the overall performance. These include the utilization of suitable cache headers in meaningful places, cache *time to live* value adjustments, resource plugin configuration adjustments and removal of unnecessary AJAX requests. Furthermore, Conmio Modules plugin could be upgraded to be capable of caching the module contents in Ehcache in a similar fashion Varnish does with ESI modules. It would further reduce the need for Varnish in the application architecture.

Finally, the origin server machines' hardware could always be upgraded by switching to more powerful machines with more CPU cores and memory. The obvious downside in

this is the increase in the server costs.

6.2 The thesis project

The thesis project was officially started in the beginning of June 2013. A few months went by searching for a suitable topic, and during the fall of 2013, the original topic was selected to be something about the Data, Context, Interaction (DCI) pattern, which is an extension to the well-known MVC pattern.

The selected topic was thoroughly studied and an artificial research problem was invented to be solved. However, during January 2014 a concrete issue in a real life second screen application scalability was encountered and improvement options were explored. During this time, it seemed sensible to change the original artificial research problem of the thesis into a concrete real life problem, even though it meant new background studies because of the new topic.

6.2.1 Successful parts

The change of the topic proved to be a good decision despite the increased workload, because as a result of the project, two useful solutions to the real life web application scalability problem were found. Because of the new topic, it was also easier to combine the actual work and the thesis work together, as both of them supported each other. Furthermore, staying in schedule became easier with the new topic than with the previous topic.

Generally speaking, the theory and technologies behind the scalability issue were not very familiar to the student before the thesis project. On one hand this increased the workload concerning the project, but on the other hand it also taught much useful information to the student; he feels more of a professional software engineer now.

When considering the original research questions set in the beginning, the thesis contains pretty thorough answers to each and every one of them, despite the fact that the performance measurements may contain some degree of error factor because of the reasons mentioned earlier.

6.2.2 Improvement areas

Although the end results of the thesis can be considered successful, everything did not go perfectly during the planning process. Because of this, it took a pretty long time to get the thesis finished, especially when the original starting point of June 2013 is considered. However, the biggest delaying factor was simply the difficulty of finding an interesting yet suitable topic for the thesis, not the writing process itself.

The performance measurements of the thesis were made with two different load testing tools, ApacheBench and JMeter. Despite they gave results, one may always criticize the

true precision of the results and measurement tools. Especially the noticeable slowing down of the JMeter simulation makes the validity of the results questionable. However, in order to utilize more advanced load testing machinery for more accurate results, an investment of some sort should have been made, which would not have been worth the money in the scope of this thesis.

REFERENCES

[1] D'souza, J. & D'souza, M. Chapter 9. Internet. In: D'souza, J. & D'souza, M (ed.). Learn Computers Step by Step. 2006, Pearson Education India. pp. 272.

[2] Arnold, J. & Becker, M. Chapter 8. Designing and Developing Mobile Internet Sites. In: Arnold, J (ed.). & Becker, M. Mobile Marketing For Dummies. Indianapolis, Indiana 2010, Wiley Publishing, Inc. pp. 384.

[3] Lauff, M. PocketWeb - WWW Browser for the Apple Newton MessagePad [WWW]. TECO - Technology for Pervasive Computing. [accessed on 3.4.2014]. Available at: http://www.teco.edu/pocketweb/.

[4] Unwired Planet Makes Internet Micro-Browser for Mobile Phones Available – Free of License Fees; Every Phone Can Now Be a Smartphone [WWW]. Gale Group. Business Wire, 27.8.1997, [accessed on 3.4.2014]. Available at: http://www.thefreelibrary.com/Unwired+Planet+Makes+Internet+Micro-Browser+for+Mobile+Phones...-a019701561.

[5] Bollen, J. WAP Formation and Philosophy [WWW]. University of San Francisco, 4.4.2000, [accessed on 2.4.2014]. Available at: http://www.usfca.edu/facstaff/morriss/651/spring00/techprojects/wap/toppage1.htm.

[6] Frederick, G. Chapter 1. Introduction to Mobile Web Development. In: Frederick, G (ed.). Beginning Smartphone Web Development: Building JavaScript, CSS, HTML and Ajax-based Applications for iPhone, Android, Palm Pre, BlackBerry, Windows Mobile, and Nokia S60. 2010, Apress. pp. 368.

[7] Mehta, N. Chapter 1. Getting Mobile. In: Mehta, N (ed.). Mobile Web Development. 2008, Packt Publishing. pp. 236.

[8] Bromby, D., Fonte, P., Forta, B., Juncker, R., Lauver, K., Mandel, R. & O'Leary, A. Chapter 9. Using WMLScript. In: Bromby, D. et al. (ed.). WAP Development with WML and WMLScript. 2000, Sams. pp. 608.

[9] Peterson, C. Chapter 1. What Is Responsive Design. In: Peterson, C. (ed.). Learning Responsive Web Design. Early release ed. 2014, O'Reilly Media, Inc. pp. 250.

[10] Enzenhofer, K., Grabner, A., Kopp, M., Pierzchala, S., Reitbauer, A. & Wilson, S. Application Performance Concepts. In: Enzenhofer, K. et al. (ed.). Java Enterprise Performance. 2012, Compuware Corporation.

[11] Dodd, A. Chapter 6. The Internet. In: Dodd, A. (ed.). The Essential Guide to Telecommunications. 5th ed. 2012, Prentice Hall. pp. 600.

[12] Henderson, C. Chapter 9. Scaling Web Applications. In: Henderson, C. (ed.). Building Scalable Web Sites. 2006, O'Reilly Media, Inc. pp. 352.

[13] Davie, B. & Peterson, L. Computer Networks. 5th ed. 2011, Morgan Kaufmann. 920 p.

[14] Rosen, R. & Shklar, L. Web Application Architecture: Principles, Protocols and Practices. 2nd ed. 2009, John Wiley & Sons. 440 p.

[15] Web Content Accessibility Guidelines [WWW]. W3C. 11.12.2008, [accessed on 30.9.2014]. Available at: http://www.w3.org/TR/WCAG20/#webpagedef.

[16] Sosinsky, B. Chapter 3. Architecture and Design. In: Sosinsky, B. (ed.). Networking Bible. 2009, John Wiley & Sons. pp. 912.

[17] Shah, S. SOA Server-Side Architecture and Code. In: Shah, S. (ed.). Web 2.0 Security: Defending Ajax, RIA, and SOA. 2007, Course Technology PTR. pp. 365.

[18] Server Configuration Options [WWW]. Oracle. [accessed on 5.9.2014]. Available at: http://docs.oracle.com/cd/E38689_01/pt853pbr0/eng/pt/tsvt/task_ServerConfigurationOptions-4d7ed6.html.

[19] W3Techs - World Wide Web Technology Surveys [WWW]. W3Techs. 2014, [accessed on 17.5.2014]. Available at: http://w3techs.com/.

[20] Hopkins, C. What is PHP? In: Hopkins, C. (ed.). Jump Start PHP. 2013, SitePoint. pp. 150.

[21] O'Dell, J. 8 Experts Break Down the Pros and Cons of Coding With PHP [WWW]. Mashable. 19.11.2010, [accessed on 18.5.2014]. Available at: http://mashable.com/2010/11/19/pros-cons-php/.

[22] Onodera, T., Suzumura, T., Tatsubori, M., Tozawa, A. & Trent, S. Performance Comparison of PHP and JSP as Server-Side Scripting Languages. IBM Tokyo Research Laboratory 2008, 19 p.

[23] Datye, V. & Kothari, N. Chapter 1. ASP.NET Overview. In: Datye, V. & Kothari, N. (ed.). Developing Microsoft® ASP.NET Server Controls and Components. 2002, Microsoft Press. pp. 722.

[24] Kohan, B. PHP vs ASP.net Comparison [WWW]. Comentum, 1.8.2010, [accessed on 19.5.2014]. Available at: http://www.comentum.com/php-vs-asp.net-comparison.html.

[25] Layka, V. Learn Java for Web Development: Modern Java Web Development. 2014, Apress. 472 p.

[26] Yegulalp, S. Surprise! Java is fastest for server-side Web apps [WWW]. InfoWorld, 11.11.2013, [accessed on 20.5.2014]. Available at: http://www.infoworld.com/t/java-programming/surprise-java-fastest-server-side-web-apps-230565.

[27] Clavijo, D. Server-side programming language statistics [WWW]. Blog Websites Frameworks, 6.3.2013, [accessed on 20.5.2014]. Available at: http://blog.websitesframeworks.com/2013/03/programming-language-statistics-in-server-side-161/.

[28] Lowmiller, J. PHP vs Java: Which is the Code for Developing a Bright Future? [WWW]. Udemy. 6.9.2013, [accessed on 14.7.2014]. Available at: http://www.udemy.com/blog/php-vs-java/.

[29] Kumar, B., Narayan, P. & Ng, T. Implementing SOA Using Java™EE. Ann Arbor, Michigan 2009, Addison-Wesley Professional. 384 p.

[30] Hales, W. Chapter 1. Client-Side Architecture. In: Hales, W. (ed.). HTML5 and JavaScript Web Apps. 2012, O'Reilly Media, Inc. pp. 172.

[31] Wasson, M. Single-Page Applications: Build Modern, Responsive Web Apps with ASP.NET [WWW]. Microsoft. 2013, [accessed on 27.8.2014]. Available at: http://msdn.microsoft.com/en-us/magazine/dn463786.aspx.

[32] Ragonha, P. JavaScript - the bad parts. In: Ragonha, P. (ed.). Jasmine JavaScript Testing. 2013, Packt Publishing. pp. 146.

[33] Framework [WWW]. DocForge. 26.11.2013, [accessed on 15.7.2014]. Available at: http://docforge.com/wiki/Framework.

[34] Grails. GoPivotal, Inc. [accessed on 15.07.2014]. Available at: http://grails.org/.

[35] Groovy [WWW]. [accessed on 15.7.2014]. Available at: http://groovy.codehaus.org/.

[36] Janssen, C. Convention Over Configuration [WWW]. Techopedia. [accessed on 17.07.2014]. Available at: http://www.techopedia.com/definition/27478/convention-over-configuration.

[37] Spring Framework [WWW]. GoPivotal, Inc. [accessed on 17.7.2014]. Available at: http://projects.spring.io/spring-framework/.

[38] Brown, J. & Rocher, G. The Definitive Guide to Grails. 2nd ed. 2009, Apress. 618 p.

[39] Koenig, D. Groovy in Action. 2nd ed. 2011, Manning Publications. 337 p.

[40] Web Framework Benchmarks [WWW]. TechEmpower. 1.5.2014, [accessed on 17.7.2014]. Available at: http://www.techempower.com/benchmarks/.

[41] Node.js [WWW]. Joyent, Inc. [accessed on 17.7.2014]. Available at: http://nodejs.org/.

[42] AngularJS [WWW]. Google. [accessed on 17.7.2014]. Available at: https://angularjs.org/.

[43] Hume, D. Part 1. Defining performance. In: Hume, D. (ed.). Fast ASP.NET Websites. 2013, Manning Publications. pp. 208.

[44] Hume, D. Part 2. General performance best practices. In: Hume, D. (ed.). Fast ASP.NET Websites. 2013, Manning Publications. pp. 208.

[45] Membrey, P., Hows, D. & Plugge, E. Practical Load Balancing: Ride the Performance Tiger. 2012, Apress. 272 p.

[46] Apache HTTP server benchmarking tool [WWW]. The Apache Software Foundation. [accessed on 17.7.2014]. Available at: http://httpd.apache.org/docs/2.2/programs/ab.html.

[47] Apache JMeter [WWW]. Apache Software Foundation. [accessed on 21.9.2014]. Available at: http://jmeter.apache.org/.

[48] Wessels, D. Web Caching. 2001, O'Reilly Media, Inc. 320 p.

[49] Google App Engine [WWW]. Google. [accessed on 10.10.2014]. Available at: https://cloud.google.com/appengine/.

[50] Varnish Cache [WWW]. Varnish Community. [accessed on 24.7.2014]. Available at: https://www.varnish-cache.org/.

[51] Nottingham, M. ESI Language Specification 1.0 [WWW]. Akamai Technologies. W3C, 2001, [accessed on 24.7.2014]. Available at: http://www.w3.org/TR/esi-lang.

[52] Jenkov, J. Servlet Filters [WWW]. [accessed on 18.9.2014]. Available at: http://tutorials.jenkov.com/java-servlets/servlet-filters.html.

[53] Ehcache: Components and Concepts [WWW]. Ehcache. 2014, [accessed on 20.10.2014]. Available at: http://ehcache.org/generated/2.9.0/html/ehc-all/#page/Ehcache_Documentation_Set/co-abt_components_and_concepts.html.

[54] Ehcache [WWW]. Ehcache. 2014, [accessed on 20.10.2014]. Available at: http://ehcache.org/.

A. EDGE SIDE INCLUDES

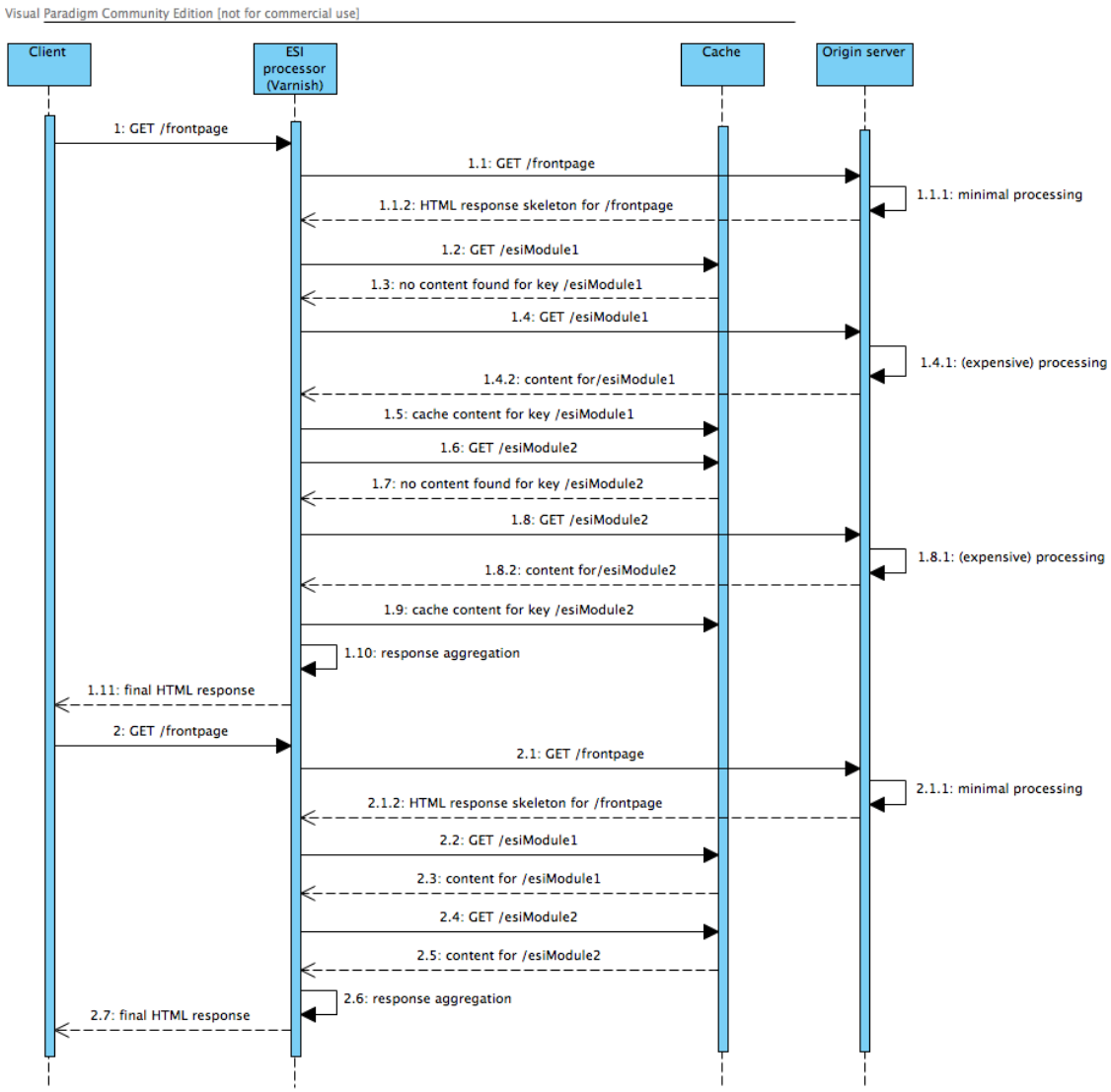

Figure A.1: The functionality of Edge Side Includes with Varnish.

B. WIREFRAMES

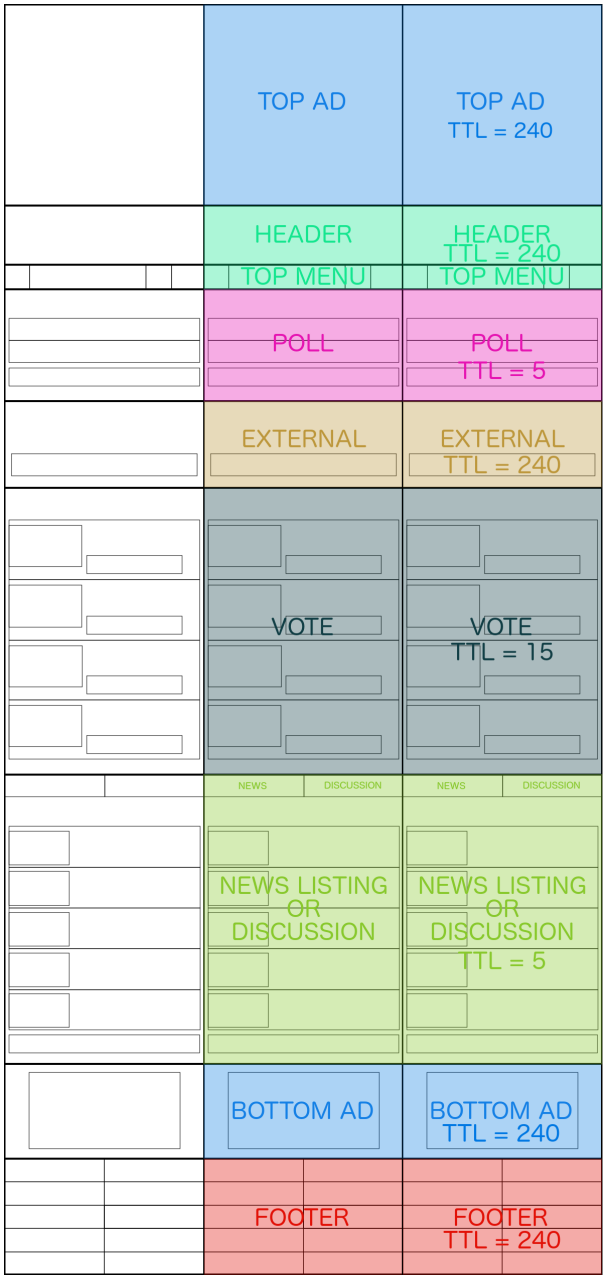

Figure B.1: The wireframe illustration of the application front page, the split into modules and the time to live values used for the modules.

C. RESPONSE CACHING FILTER

```groovy
class ResponseCachingFilter extends SimplePageCachingFilter {

    def webApplicationContext
    def esiVariableService

    private static final List<String> shouldBeFilteredPaths = [
            "/frontpage",
    ]

    @Override
    public void doInit(FilterConfig filterConfig) {
        super.doInit(filterConfig)
        webApplicationContext = WebApplicationContextUtils.getWebApplicationContext(filterConfig.getServletContext())
        esiVariableService = webApplicationContext.getBean("variableService")
    }

    @Override
    protected void doFilter(HttpServletRequest request, HttpServletResponse response, FilterChain chain) {
        String requestURI = request.getRequestURI()
        if(shouldBeFilteredPaths.find{String path -> requestURI.startsWith(path)}) {
            super.doFilter(request, response, chain)
        }
        else {
            chain.doFilter(request, response)
        }
    }

    @Override
    protected String calculateKey(HttpServletRequest request) {
        StringBuilder stringBuilder = new StringBuilder()
        stringBuilder.append(request.getMethod()).append(request.getRequestURI()).append(request.getQueryString())
        stringBuilder.append(getVariableStringForKey(request))
        return stringBuilder.toString()
    }

    private String getVariableStringForKey(HttpServletRequest request) {
        StringBuilder stringBuilder = new StringBuilder()
        String requestURI = request.getRequestURI()
        Map variables = esiVariableService.getMenuVariables(request)
        variables.putAll(esiVariableService.getFrontPageVariables(request))
        variables.each{String key, def value ->
            stringBuilder.append("${key}=${value?.toString()}|")
        }
        return stringBuilder.toString()
    }
}
```

Figure C.1: The required code for overriding and extending the SimplePageCachingFilter implementation in order to cache the application front page responses.

D. BENCHMARK RESULTS

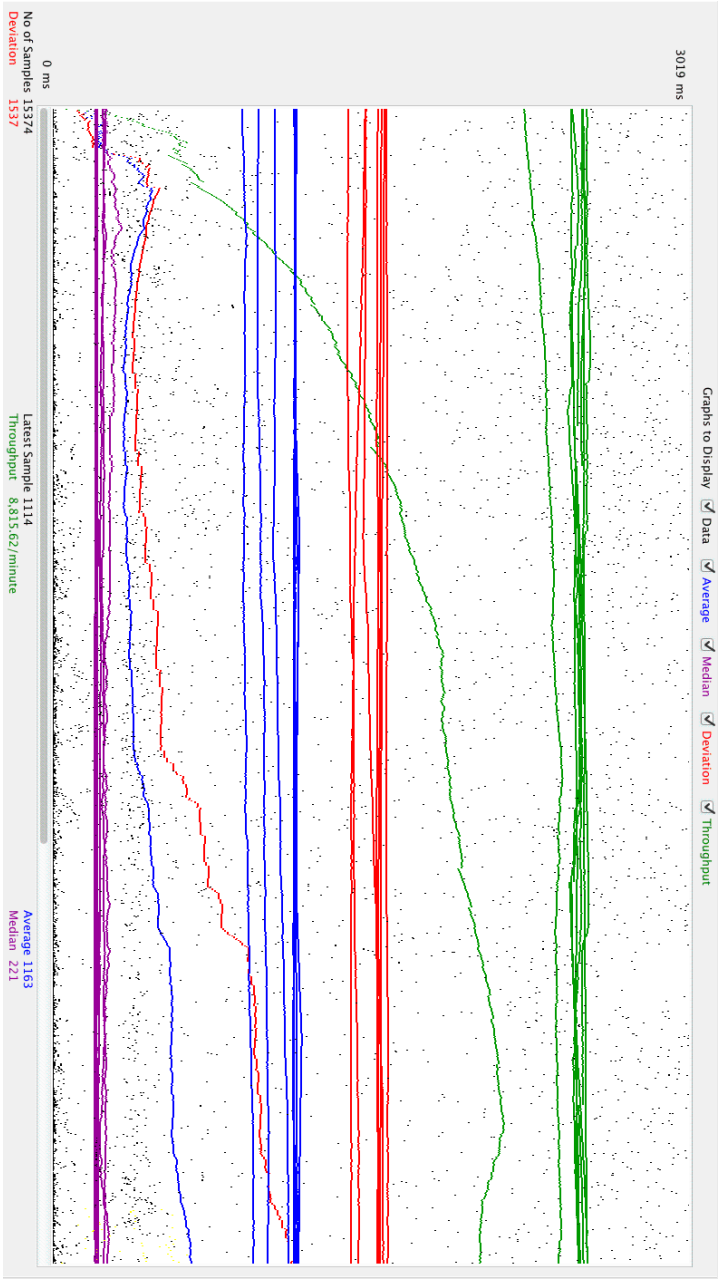

Figure D.1: The general results of the original application JMeter simulation measured through Varnish.

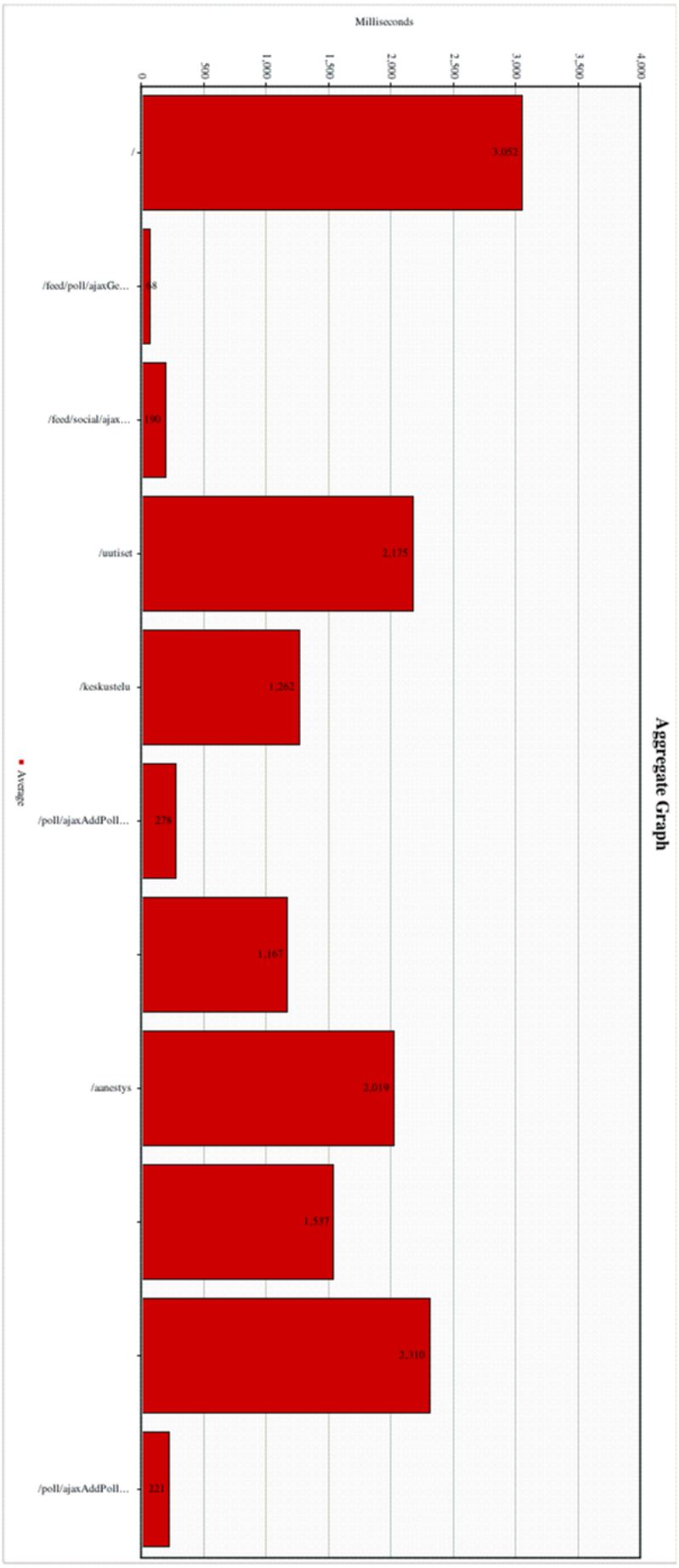

Figure D.2: The average response times of different requests of the original application measured through Varnish.

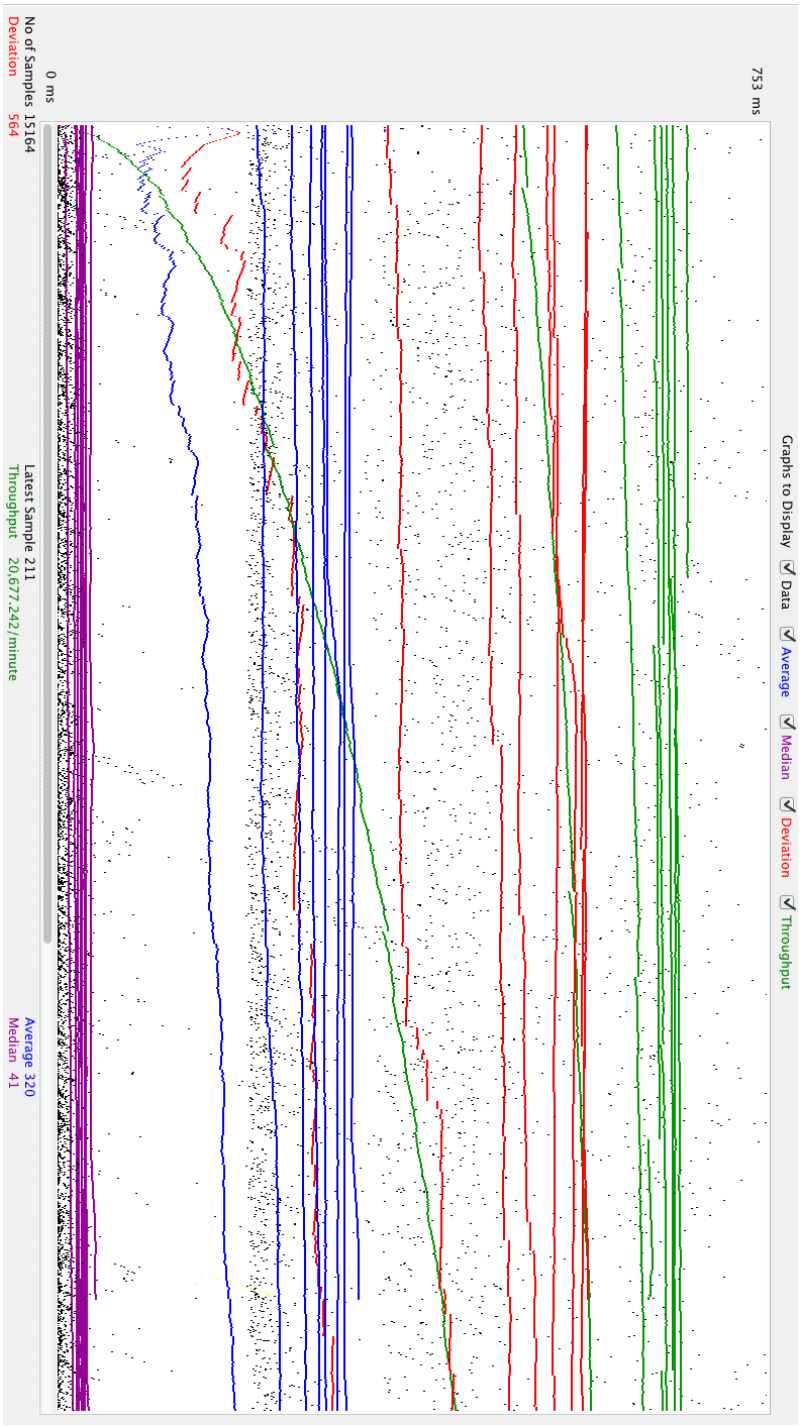

Figure D.3: The general results of the optimized application JMeter simulation measured through Varnish.

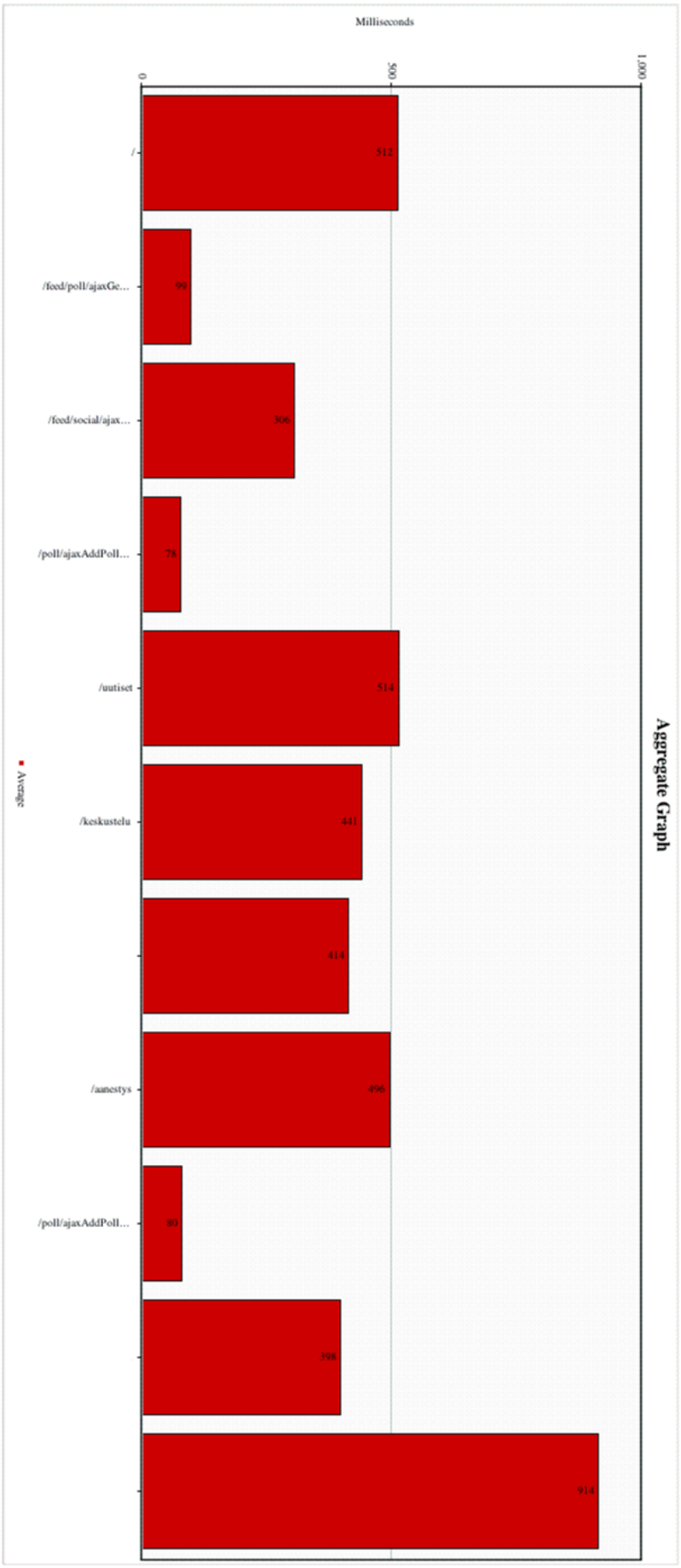

Figure D.4: The average response times of different requests of the optimized application measured through Varnish.

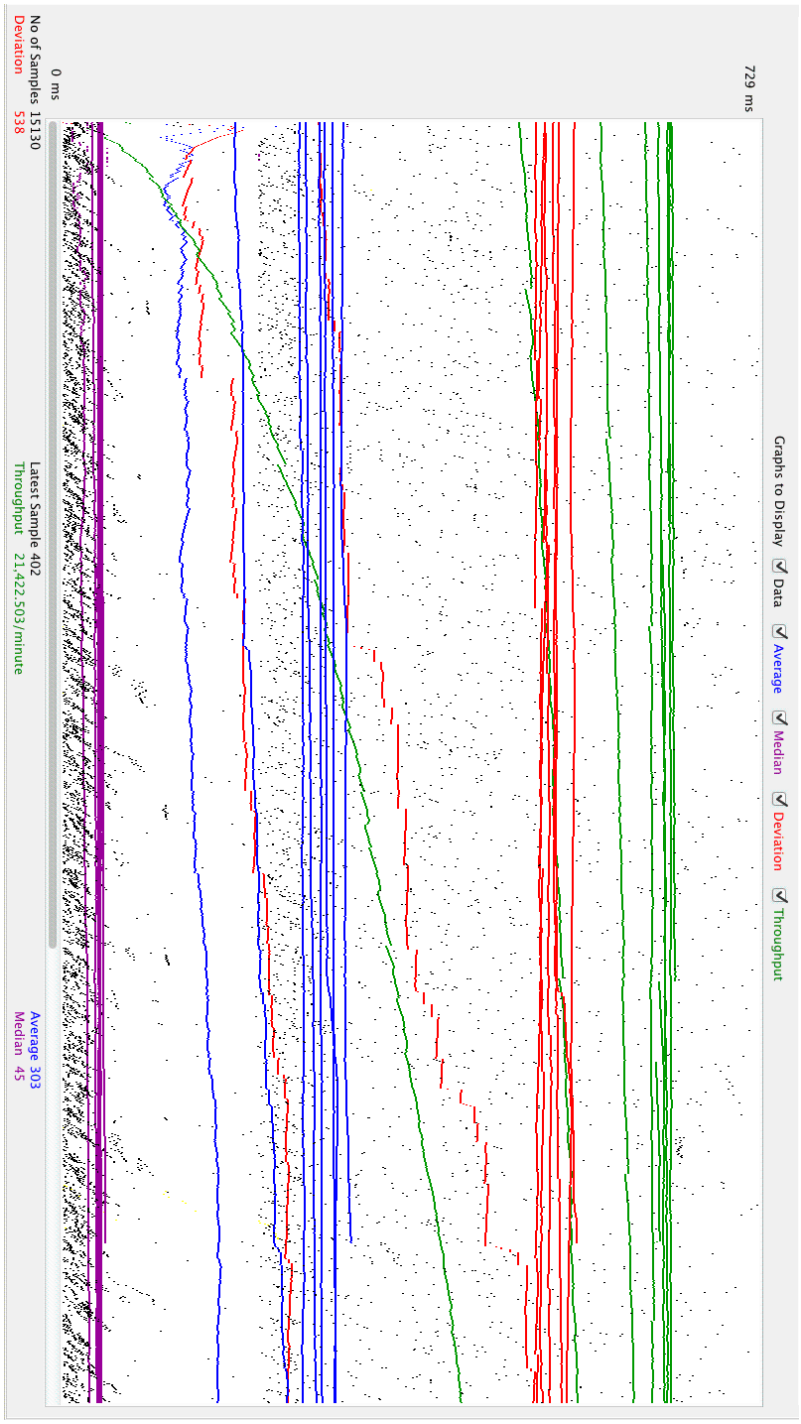

Figure D.5: The general results of the optimized application JMeter simulation measured through Apache HTTP server.

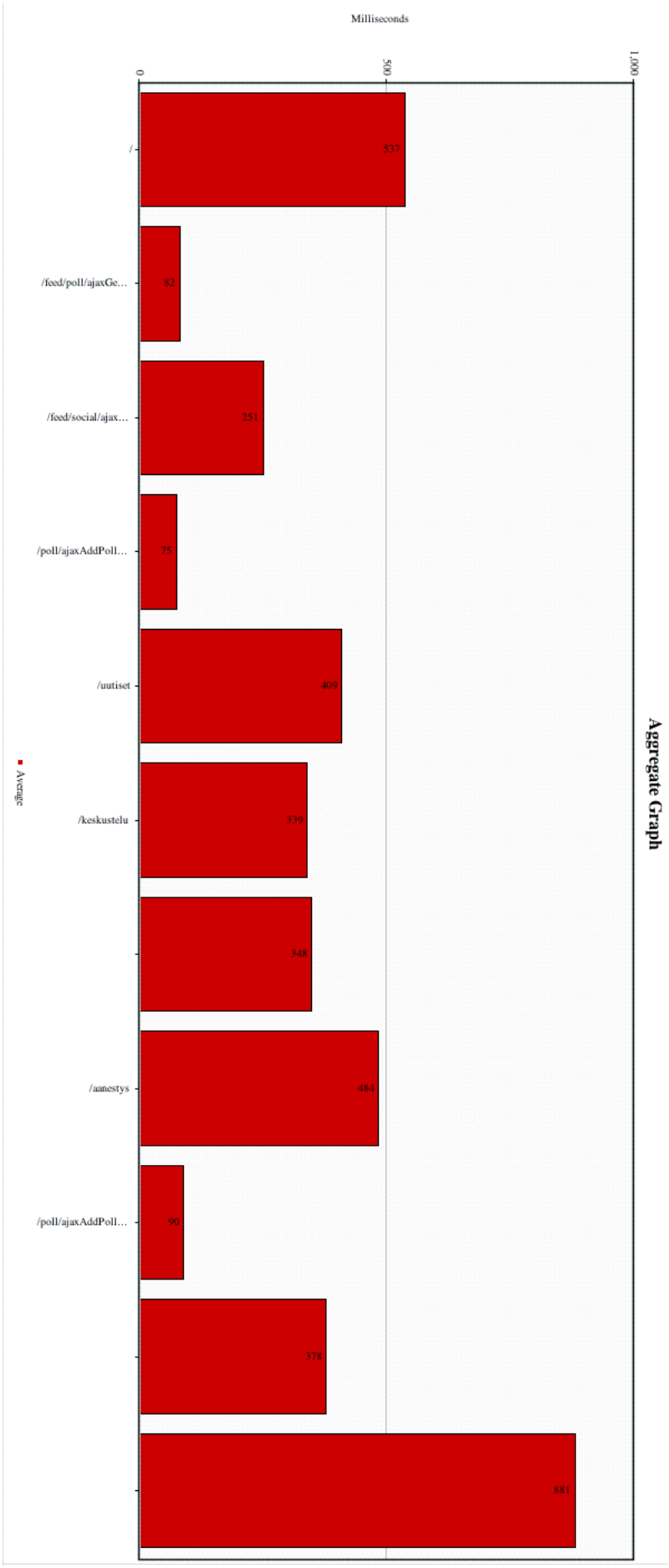

Figure D.6: The average response times of different requests of the optimized application measured through Apache HTTP server.