Faculty of Science and Technology

MASTER'S THESIS

Study program/ Specialization: Computer Science	Spring semester, 2016 Open / ~~Restricted access~~
Writer: Erik Solhaug	 (Writer's signature)
Faculty supervisor: Reggie Davidrajuh External supervisor(s): Nejm Saadallah	
Thesis title: Design and implementation of a real-time web based interactive system	
Credits (ECTS): 30	
Key words: Real-time transportation WebSockets Single-page Application REST Multitier architecture	Pages: 53 + enclosure: Appendix & attached compressed file containing source code Stavanger, 15/06/2016 Date/year

Acknowledgments

I would first like to thank my thesis advisor, Professor Reggie Davidrajuh at the University of Stavanger. Professor Davidrajuh has given me great moral support, inspiration and helped me get up when my motivation was down. He has given me good guidance in terms of writing and structure and has helped me with forming this thesis.

I would also like to thank and express my sincere gratitude to my external thesis advisor, Nejm Saadallah at the International Research Institute of Stavanger. Nejm has been a great support during this period, and has challenged me in an academical and technical manner with his wide knowledge. It has helped me form this thesis to be my own work. He has always had an open door policy, which gave me the confidence to reach out for help in the times when it was needed the most. I am grateful for being able to send email at any hour of the day and getting a response shortly after. Good luck with your guitar lessons.

I am grateful to International Research Institute of Stavanger for providing me with office space and equipment to carry out my research. I want to thank Sonja Moi for guidance and Robert Ewald for helping me with specific problems that occured during this period and giving directions to the scientific thought process.

I would like to thank Peder Thorup, which at the time of submitting this thesis, is also writing his Master's thesis in Computer Science at the University of Oslo. Peder has helped me as the second reader of this thesis. I am grateful for the valuable comments, input and support he has given me throughout this period. When he is to submit his thesis, I will certainly reciprocate the help he has given with the same service.

I want to thank my fellow students of the masters program in Computer Science at the University of Stavanger, for making the last two years of this masters degree one of the greatest experiences I have had. Thank you for all the late night studying, stimulating and non-stimulating discussions and the fun we have experienced together.

Finally, I must express my profound gratitude to my family for providing me with unfailing support and continuous encouragement throughout my years of study and through the process of researching and writing this thesis. Without them, this accomplishment would not have been possible.

Thank you.

Stavanger, 15/06/2016

Abstract

This thesis explores the concepts of real-time interactive web based systems. The International Research Institute of Stavanger wants to deploy their complex stand-alone simulators to a web application where clients can interact with the simulator over the Internet. We propose a general purpose multitier architecture addresses this problem. We research the underlying technologies, methods and paradigms that are core to the problem statement with a hands-on approach. We look into the concepts of building interactive web applications known as Single-page Applications and utilizing real-time transportation of data with the help of WebSockets. We have made a prototype based on the proposed architecture, in an attempt to address underlying problem statements which include scalability, reliability and quality of service. The prototype has been tested through emulating user interaction which has produced a set of results. We conclude around the results with discussion about finding the root cause or to react symptomatically.

Acronyms and abbreviations

UiS University of Stavanger
IRIS Internation Research Institute of Stavanger
HTTP HyperText Transfer Protocol
HTML HyperText Markup Language
CSS Cascading Style Sheet
RPC Remote Procedure Call
JS Javascript
SOAP Simple Object Access Protocol
SMTP Simple Mail Transfer Protocol
JSON Javascript Object Notation
REST Representational State Transfer
XML Extensible Markup Language
URI Uniform Resource Identifier
URL Uniform Resource Locator
API Application Programming Interface
RTT Round Trip Time
TCP Transmission Control Protocol
IP Internet Protocol
AJAX Asynchronous Javascript and XML
DoS Denial of Service
CPU Central Processing Unit
RAM Random Access Memory
OSI Open Systems Interconnection Basic Reference Model
CRUD Create, Read, Update, Delete
SPA Single-page Application
I/O Input/Output
IIS Internet Information Services
RTSys Real-time System

Contents

List of Figures

List of Examples

List of Tables

Chapter 1

Introduction

A web based system is a system that is based on web browsers receiving data and interact with content located on documents called web pages. When addressing web pages as documents, they may be perceived as static presentations of content. This was the case from the early 1990's and back to their origin. Today, modern web sites are highly dynamic and are often referred to as web applications. Web browsers work as an operating system for the web applications that are being displayed for millions of users every day and is to an extent featured as computer programs. In traditional web pages, the client retrieves information from the server, and is required to wait for all content to load, which happens in a synchronous manner. In the modern day web applications, the pages are usually loaded as a "template", and the content is loaded asynchronously "behind the scenes" from the server, also known as Single-page Applications. This gives the user a experience similar to using native application, making it interactive.

Real-time is a known concept, and in computation it has a different meaning than it does in transport. This thesis will not consider the concept of real-time in terms of computing, but in terms of transport. When considering the term transport, we will be measuring the round-trip time. The expected response time of an action may vary, but for an average human being it is thought to be in the vicinity of [250] milliseconds from action to response to be considered real-time. An analogy for this would be pushing the stop-button on the bus, and the stop-light turns red within fractions of a second. The web in its intended design, is not meant to handle communication in real-time. Over the years, there have appeared options to deliver data in a real-time fashion, such as WebSockets.

This thesis will explore the different concepts around an architecture that embraces a real-time communication protocol with a modern day interactive web application. There will be an exploratory hands-on approach to see and discuss if the implementation of a proposed architecture is viable to achieve a goal of being a reliable real-time web based system in a modern day interactive web application.

1.1 Problem statements

The problem statements are as follows:

1. **What steps are necessary to make a modern web application interactive?**

Web applications are traditionally a set of static documents which are being sent from a web server when you request them, which is a synchronous action. With the innovative development of Javascript the past twenty years, how is it possible to make a web application interactive with the use of asynchronous requests?

2. **How will the proposed architecture for this project scale?**

An architecture and its setup tells us how it is expected to scale. How will the proposed architecture scale in terms of horizontal and vertical scaling?

3. **Are WebSockets or WebSocket based libraries mature enough to handle reliable transportation of data?**

WebSocket is not supported by every browser, and there are no re-send mechanisms of lost packets when the connection drops included in the protocol. Is there a way to ensure that data is being delivered correctly without resulting in the service haulting or stopping? We will explore and see if there are libraries that assist in reconnection or fallback to alternative transport protocols.

4. **Will multiple persistent connections result in a deterioration of the quality of service the system will provide?**

When dealing with many clients connected to a service, there is an expectation that the performance will suffer to some extent when it comes to performance and data delivery. We will do stress-testing of the implemented solution to monitor the communication of clients and server with low, medium and high load in addition to low and high message exchange rate in regards to the load ranges.

1.2 Related work

There have been written papers on related matters earlier, such as Johannessen, Kristian[1], where Johannessen compares different frameworks for real-time on the web in his masters thesis. He also compares HTTP techniques against WebSocket. Johannessen concludes with *"WebSockets is superior to HTTP when it comes to bi-directional communication"* and after testing various frameworks, such as Lightstreamer[2], SignalR[3] and Socket.IO[4], that SignalR is the best framework for real-time web, with Socket.IO as a close runner-up.

From the same university as Johannessen, University of Oslo, one year later, Øyvind Raasholm Tangen writes about Real-Time Web with WebSocket.[5] Tangen focuses on the performance of WebSockets and other real-time transports provide in terms of load on CPU and memory. Tangen concludes with *"Under moderate server CPU load levels, WebSocket is recommended over Server-Sent Events and HTTP Long Polling, both in terms of performance and programmer friendliness. However, under extreme levels of server CPU load, HTTP Long Polling is the preferred transport."*. Tangen reports that when the server reaches approximately 300 clients, the CPU load reaches 100 %, which results in increasing response times.

1.3 Thesis outline

This thesis consists of six parts, divided in chapters. These parts intend to make the context of the thesis apparent to the reader. All chapters are divided in sections consisting of subchapters and topics.

Chapter 1: Introduction: The introduction gives an overview over the thesis, problem statements and the related works.

Chapter 2: Technological context: This chapter describes and discusses relevant technologies has been explored and used with regard to the thesis. The chapter provides a historical context to the development of the relevant technologies.

Chapter 3: Architecture: This chapter describes the design of the architecture as modules. The chapter will explain concepts based on the findings in the second chapter and their role in the architecture.

Chapter 4: Implementation: The implementation chapter provides information about which parts of the proposed architecture are implemented.

Chapter 5: Test and Results: This chapter is divided in two parts; the first part provides information about how to deploy the server and client, and what is needed to get started. It contains a sample run of the implementation on both client and server side with screen shots attached. The second part is a stress-test of the solution where we display performance graphs and statistical values.

Chapter 6: Discussion: The last chapter gives a description of the originality of the works, what limitations are present, future work, learning experience of this project, results and conclusion.

Chapter 2

Technological context

When we are looking at technological context, it may be useful to an overview of which elements are required to reach the goal of the problems. Figure 2.1 presents a mindmap of the different areas that we focus on.

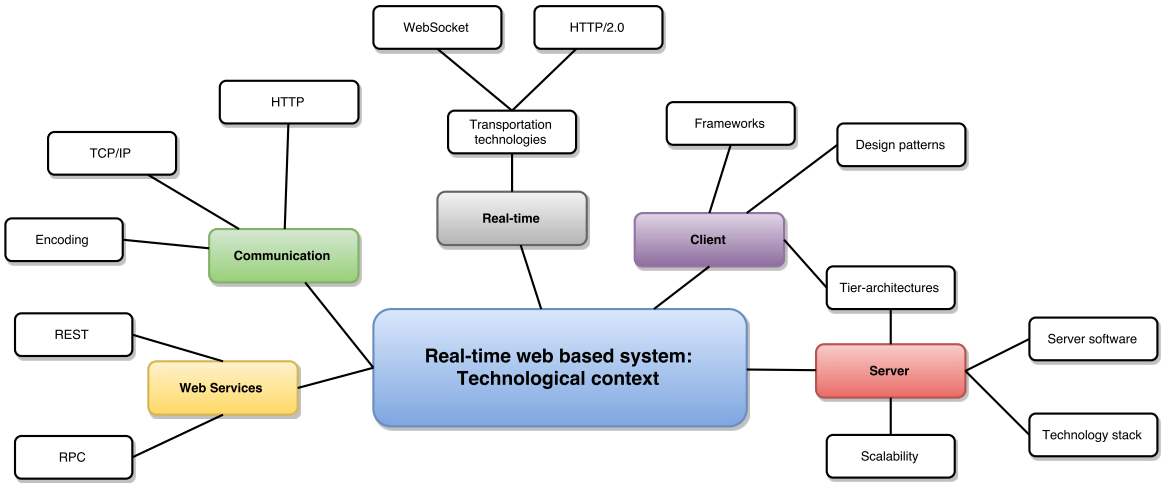

Figure 2.1: Mindmap: Technological context

2.1 Context

The problem that we want to address in this thesis is real-time interaction with modern day web application that feels almost like a desktop application. The interaction is to be with a simulator, which in this thesis is a generic term for a content generator which role is to generate the next state based on current state and user input. This data will then be sent to the client. The interaction with the application and the data transfer must be of a nature that ensures fast response and delivery. To achieve this, we have explored a set of web communication technologies, web architectures and other technologies that may help to solve this problem.

When discussing communication technologies that revolves around the web, it is usually around the TCP/IP-model and the protocols and services that are being carried out over the TCP-protocol. The TCP/IP-model consists of four layers; *Network Interface layer*, *Internet layer*,

Transport layer and the *Application layer*. The model resembles the OSI-model[6], but consists of fewer defined layers as illustrated in Figure 2.2.

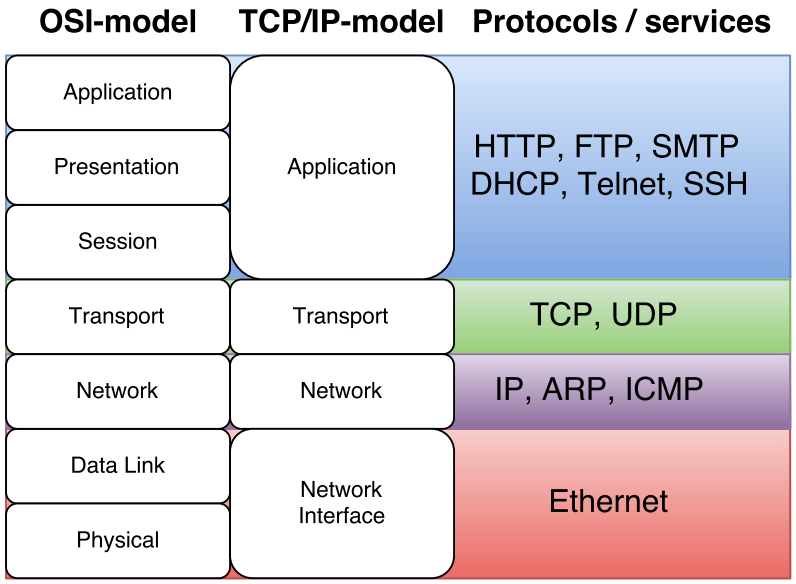

Figure 2.2: Communication models: OSI, TCP/IP

In the architecture and implementation, the focus is on technology that spans the transport layer and the application layer. The communication, in terms of information and content, that is not regarding real-time data to and from the client, is carried out by the HTTP over TCP. This communication is a directed client-server request-response communication as illustrated in Figure 2.3. The figure illustrates the communication between a client and a server, and the exchange of HTML-documents and other documents that are attached to the display of the content requested. A traditional web application does these requests every time they perform an action on the web application that requires change in the content. The HTTP protocol is stateless, meaning that the client and server do not have any information about each other. This means that the server do not need to allocate resources to deal with a client but only serve what is needed at that particular moment the request is made.

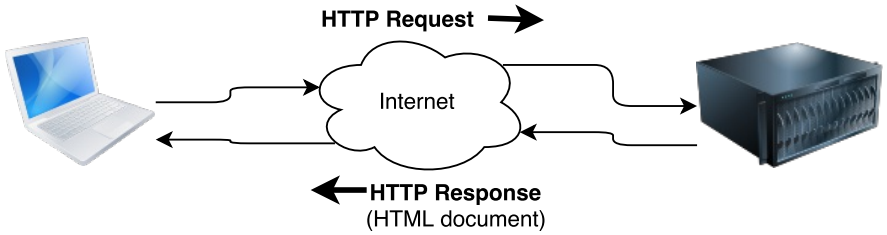

Figure 2.3: HTTP: client-server request-response communication

A traditional web application is synchronous, meaning it has to wait until the document it has requested has been received and rendered. Modern day applications handles this process without rendering the entire view for each request, but they will asynchronously load parts

of the view they need with the help of Javascript through a service called Web services, which we will cover later in this chapter. In addition to directed communication, we also have bi-directional communication. A bi-directional connection establishes a direct link between the client and a server through a technology called WebSockets. This is a persistent, stateful connection, meaning the client and server has allocated some resources to communicate at any given time with each other until one of them chooses to close the connection. Both directional and bi-directional communication are technology concepts that we are going to explore and implement.

2.1.1 The request-response paradigm

As we mentioned the web is built around the request-response paradigm of HTTP. As the Internet has gained traction and the performance of computers is increasing, users are losing patience faster when dealing with systems that use some time to load, Jakob Nielsen wrote in 1991 about the three important limits to response times[7]: "if an user experiences delays under 1.0 second, the users flow of thought is to stay uninterrupted." Imagine having a web application that has a lot of content, such as text, images, video and other media. When navigating these pages, the server will send the whole document to the client alongside the media that is attached. If the bandwidth and speed of the given users Internet is limited, it is likely that the user will lose interest in the web application. The key to these kind of web application was to load content without re-rendering the whole page. But how could this be done? In 1996, Microsoft introduced the iframe tag for HTML, which is used to display an inline frame to the view with content that can be retrieved from an external source. This goal with iframe was to retrieve content asynchronously. This technique later developed further in to XMLHTTP[8] through an ActiveX plug in[9] for Internet Explorer. In 2005, Asynchronous Javascript and XML (AJAX) was designed, which uses a set of techniques to retrieve and load content to the user without re-rendering the whole web application which provided unnecessary additional frames provided inside the web application or plug ins.

With the birth of AJAX, new techniques emerged that embraced asynchronous request-response. We can now poll the server to ask for updates in content. This technique is called Ajax Polling and works by sending requests to the server asynchronously at given time intervals. Another variation of this technique is called AJAX Long-Polling, which leaves open a larger request window for the server to respond when it has new information.

2.1.2 Web services and WebSockets

As this project is utilizing communication methods that consists of both directional and bi-directional communication, it is natural to provide some context of Web services and WebSockets which embraces these communication types. This chapter explores the historical development of both Web services and WebSockets and development of client and server structure and architectures in regards to the implementation of this project.

Web services: Remote Procedure Calls

Remote Procedure Calls, also known as RPCs, is a protocol that is used to request information that lies on a remote machine on a remote network and was mentioned in research articles as early as 1976.[10].

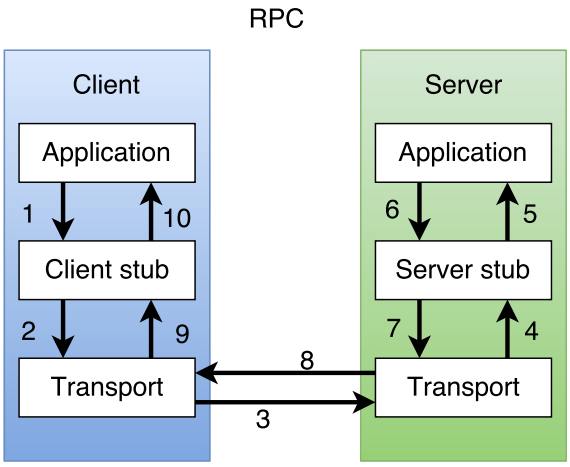

Figure 2.4: RPC communication

The explanation of Figure 2.4 is better explained by Microsoft: *"The RPC process starts on the client side. The client application calls a local stub procedure instead of code implementing the procedure. Stubs are compiled and linked with the client application during development. Instead of containing code that implements the remote procedure, the client stub code retrieves the required parameters from the client address space and delivers them to the client runtime library. The client runtime library then translates the parameters as needed into a standard Network Data Representation (NDR) format for transmission to the server."*[11]

In the early days, to use RPC, you were required to know the infrastructure of the destination *application* you were trying to reach. There was no standardized way of exposing the host API and no structured way of sending and retrieving the information. Fast forward twenty years, and there has been several attempts to create standardized ways of retrieving information through RPC, when there finally was an agreed upon solution presented by Microsoft called SOAP[12]. SOAP stands for Simple Object Access Protocol, and is an XML-based protocol for message exchange through remote procedure calls and is language independent. This protocol was intended to work on existing transports, such as HTTP and was originally designed for a distributed computing environment. The protocol is an envelope which consists of two elements; header and body. An example of this can be seen in Example 2.1.
SOAP provides a structure that is readable for both humans and computers through XML, and is easily verifiable, but provides a lot of overhead due to its structure and verboseness. The XML-format needs to be parsed which results in extra computation, thus adding a delay in the communication process. SOAP that uses Web Service Definition Language (WSDL), contains information about how the API of the host is to be used. In a short summary, the WSDL contains an abstract of definitions that tells what kind of request and response types are present and messages describing methods the host are using. It also tells what the expected responses are to the requests that are being issued[14].

```
POST /travelservice
SOAPAction: ‘‘http://www.acme-travel.com/checkin’’
Content-Type: text/xml; charset=‘‘utf-8’’
Content-Length: nnnn

<SOAP:Envelope xmlns:SOAP=‘‘http://schemas.xmlsoap.org/soap/envelope/’’>
    <SOAP:Body>
        <et:eTicket xmlns:et=‘‘http://www.acme-travel.com/eticket/schema’’>
        <et:passengerName first=‘‘Joe’’ last=‘‘Smith’’/>
        <et:flightInfo airlineName=‘‘AA’’
                       flightNumber=‘‘1111’’
                       departureDate=‘‘2002-01-01’’
                       departureTime=‘‘1905’’/>
        </et:eTicket>
    </SOAP:Body>
</SOAP:Envelope>
```

Example 2.1: Example SOAP-structure. Source: Unraveling the Web Services Web[13]

Web services: REST

In addition to RPC and SOAP, there are other ways to communicate with other hosts, and one of those ways is to use point-to-point communication. REST stands for **Re**presentational **S**tate **T**ransfer and is a software architecture that was made apparent by Roy Thomas Fieldings PhD dissertation[15] that allows communication between machines without the use of mechanisms other than HTTP. REST in it self is not a Web service, but in this thesis we address it as such for the sake of simplicity and consistency. Operations are carried out over HTTP using Uniform Resource Identifier (URI), with HTTP methods[1] [16]. These methods are descriptive and contains enough information for the server to process the message or command that is being executed. When dealing with operations with persistent storage in traditional computer programming, we often refer them as CRUD-operations. These operations consists of *create, read, update* and *delete*. The table below shows CRUD-operations and their HTTP verb-equivalents:

CRUD	**HTTP**
Read	GET
Create, update, delete	POST
Create, update	PUT
Delete	DELETE

Table 2.1: CRUD and HTTP

In the same way we use CRUD-operations in traditional computer programming, we use the HTTP verbs to achieve the same results through REST. Figure 2.5 shows a simplified communication between client and server, where the client sends a GET-verb to the server on a given

[1] In the context of discussing REST, HTTP methods are more known as "HTTP verbs"

URI. The server will then typically respond with a JSON- or XML-encoded message that represents the information that is being requested[17].

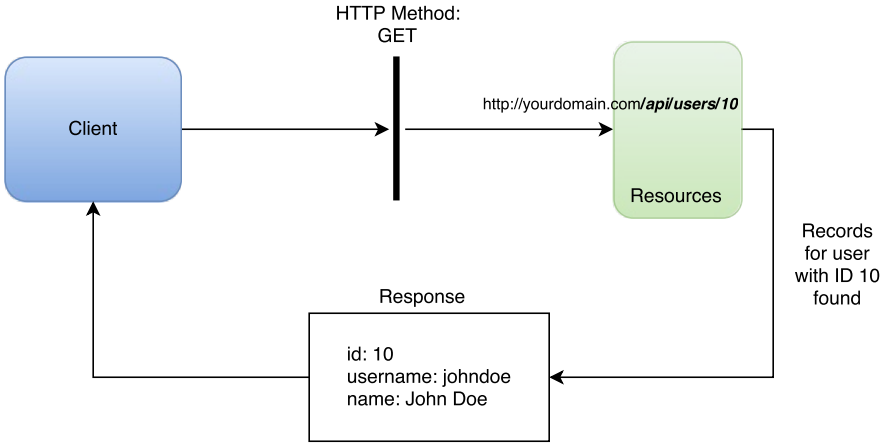

Figure 2.5: Simplified illustration of information retrieval with GET (REST)

The URI illustrated in the figure contains an example-setup of a server with a RESTful API; **/api/users/10**, where Table 2.2 shows examples of operations that can be done.

API	**Table**	**Value**	**HTTP verb**	**Action**	**Result**
api/	users/	10	GET	Get information with id 10	Example 2.4
api/	users/	10	PUT	Update entry with id 10	Updated with new values
api/	users/	11	POST	Insert entry on id 11	New user entry
api/	users/	11	DELETE	Delete entry on id 11	User deleted

Table 2.2: Properties of a REST URI

If we look at the first example of Table 2.2, the URI might also look differently, in the likes of **/api/?table=users&id=10**, where you query the server with parameter **table** and value **users**, and parameter **id** with value **10** with GET[17]. The client-server communication is typically illustrated with Example 2.2, 2.3 and 2.4.

When dealing with a REST API, there usually is no description of how to interact with the Web service unless there has been one specified. The service description is not mandatory to include with a REST service as it is with SOAP, but if it is chosen to be included, it is called a Web Application Description Language (WADL)[18]. REST is frequently used in modern frameworks and implementations of modern day web applications. For that reason, REST is the Web service we want to explore further by implementing it in the proposed architecture.

```
GET /api/users/10 HTTP/1.1
Host: yourdomain.com
Content-Type: application/json
Cache-Control: no-cache
```

Example 2.2: Raw headers being sent to the server

```
// GET api/users/<id>
public User Get(int id) {
    var user = getUserByID(id);
    return Json(user, JsonRequestBehavior.AllowGet);
}
```

Example 2.3: Server side: Pseudo code of processing the GET-request in C#

```
{
    "id": 10,
    "username" : "johndoe",
    "name" : "John Doe"
}
```

Example 2.4: Client side: JSON-response from the server

Real-time: WebSockets

As mentioned earlier in this chapter, we are exploring communication technologies that spans the transport layer and the application layer. Socket programming is not revolutionary or new, it has existed for quite some time. If you have accessed servers through Telnet or SSH, you have used programs that utilize sockets. WebSockets is a type of socket and is one of the newer protocols that has been introduced to the world of web communication. WebSockets have no relationship to HTTP besides the handshake and Upgrade-request instantiated by the client to the server, so a WebSocket connection can be instantiated in parallell to the server. An illustration of this can be found in Figure 2.6. This process is initiated with a HTTP-upgrade request for WebSocket connection from the client to the server as illustrated in Example 2.5. Given that the server has implemented WebSocket support it will respond with an agree-message as shown in Example 2.6. An illustration of this can be found in Figure 2.6. WebSocket is an independent TCP-based protocol which is implemented with the URI-scheme *ws://* or over Transport Layer Security (TLS) with the URI-scheme *wss://* that provides secure information transfer[19].

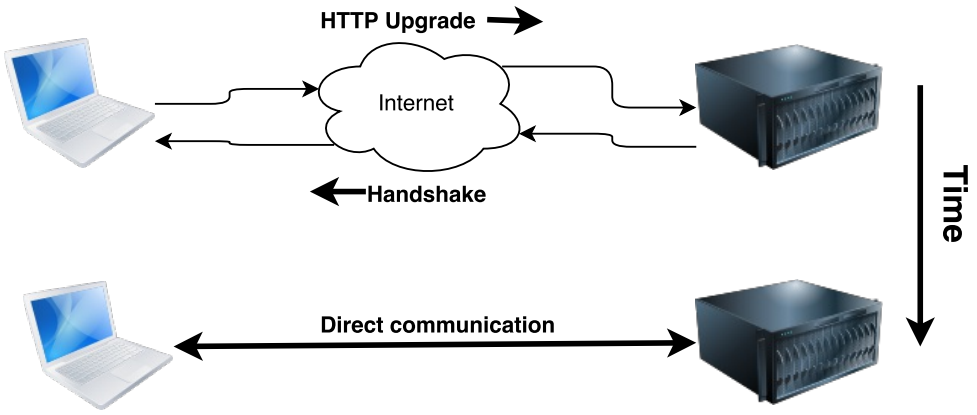

Figure 2.6: Bi-directional communication between client and server

WebSockets is the protocol used on the web to handle real-time communication. The protocol establishes a persistent connection between client and server and allows the client and server to push information to each other without sending requests in real-time. As WebSockets have aspired in the real-time web development community and provides a lot of documentation and support, we will explore it further by implementing it in the proposed architecture.

```
GET /websocketservice HTTP/1.1
Host: yourdomain.com:8000
Upgrade: websocket
Connection: Upgrade
```

Example 2.5: WebSockets HTTP Header: Client handshake request for WebSocket upgrade

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
```

Example 2.6: WebSockets HTTP Header: Server handshake response for WebSocket upgrade

HTTP/2.0

HTTP/1.1 was introduced in 1999, which from the date of writing this thesis, is 17 years ago. Given that web technology, frameworks and paradigms have evolved, HTTP has not. Web applications are getting more complex every day, and have other demands than 20 years ago. HTTP/2.0 is the next revision of HTTP in the making, and is being maintained by the Internet Engineering Task Force (IETF). The revised protocol is to contain the same HTTP verbs, status codes, semantics and API as the already well incorporated HTTP/1.1, but will be a binary protocol instead of text based. It will also provide some other interesting aspects. For instance, when

requesting a web page to be rendered, we have to ask the server for the HTML document, the Javascript dependencies that are being used, the stylesheets (CSS-files) and other media that the rendered page is to display. This requires multiple GET-requests to the server, meaning several HTTP-requests over several TCP connections. The new protocol supports multiplexing several requests into a single TCP-connection, reducing the overhead and the overall latency and providing a bi-directional communication channel[20]. For a time, it was believed that HTTP/2.0 was going to replace WebSockets, because of the bi-directional channel and the server-push mechanism it provides. However, the server-push may only push new content to the clients cache, and not directly to its view. It is not unthinkable to believe that HTTP/2.0 will be used for small-scale trivial data in real-time transportation.

2.1.3 Architectural context

Now that we have an overview of the communication context, we will explore some different types of architectures and present the different layers that are being used in software in general and for web applications. These architecture types gives the background for the proposed architecture.

One-Tier architecture

A one-tier architecture is an architecture that keeps all the elements that are a part of an application or solution in one place and the communication is one-sided. These types of systems are very simple and are often referred to as stand-alone applications or a single process as illustrated in Figure 2.7 and are often placed in isolation with no possibility to be connected to besides in the immediate local area or network.[21]

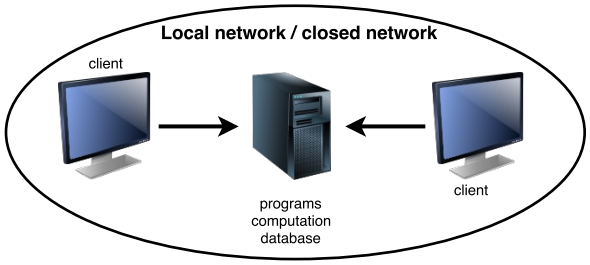

Figure 2.7: One-Tier architecture: System in Local network / Closed network

Two-Tier architecture

The next step after one-tier architecture, is the two-tier architecture. It has a client-server relationship and the direct communication takes place between the two entities in a one to one fashion. The two entities can be described as the client application and the server which acts as a data source. Users using the client have the opportunity to directly manipulate the data source through their application remotely. Two-tier architectures are often implemented as user interfaces that directly access databases. An illustration of two-tier architecture can be found in Figure 2.8[22]

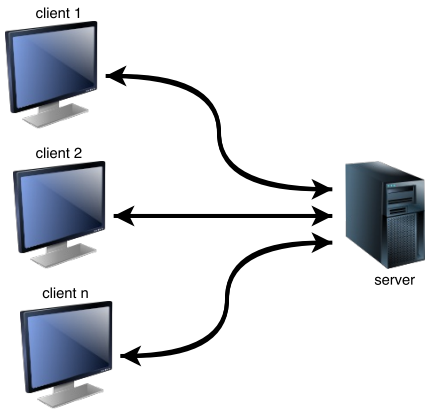

Figure 2.8: Two-Tier architecture: Client-server system

Three-Tier architecture

By segmenting the server from the two-tier architecture, we can divide the server and the data source to act as two independent entities. As illustrated in Figure 2.9, we now have three entities, which also is referred to as "layers". These layers consists of a *Presentation layer*, *Business logic layer* and *Data access layer*. These three layers are very often used in web development. The Presentation layer consists of the user interface the user is presented with. In this layer, the user can input data or get data output. The input data is transferred over to the Business logic Layer, where logical operations of data is being done. This layer takes care of calculations, validation of data and filtering before sending it further along the architecture to the Data access layer. The Data access layer now performs the CRUD operations that the user has requested and the business logic layer has operated.[23]

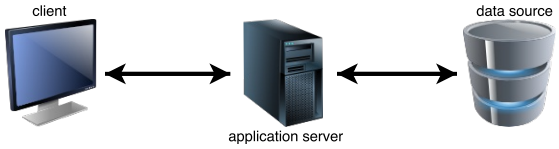

Figure 2.9: Three-Tier architecture: Client, server and data storage

Multitier architecture

A multitiered architecture is in many ways very similar to a three-tier architecture. The main difference is that the middle tier which consists of the Logic layer, which itself consist of several modules or tiers. What is considered tiers is debatable, but for the simplicity, we refer to modules that consist within the Logic layer as tiers. In web development, these modules are referred to as middleware. A multitier architecture is also commonly known as *n-tiered*.[24] This architecture is the type we are proposing in this thesis, and will look into the different modules related to the problem statements. An illustration of a multitiered architecture can be found in Figure 2.10.

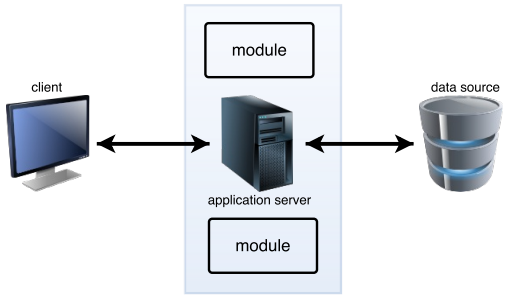

Figure 2.10: Multitier architecture: Three-Tier Architecture with modules

Types of architectures

There are two terms when it comes to architectures, which are **thick** and thin architectures. A **thick** architecture is software which usually provides rich functionality and all logic is executed in one place, such as a web server or a client. A **thin** architecture will do minimal computation. If a web server is thin, the client will have to do the computation and vice versa.

Scalability

When we are discussing scalability of architectures, there are two terms:
Horizontal scaling: If an architecture with a cluster needs resources to improve performance, you can scale it with adding more machines. Effectively, this means that you can add inexpensive machines to the machine pool or cluster to expand. The benefits is that you can continue to add machines indefinitely, but physical space will become an issue. Horizontal scaling might be good for an architecture that is in need of more load balancing than computation.
Vertical scaling: If an architecture needs more resources to improve performance, you can scale it with adding more hardware to the machine that needs improvement. With the right hardware, you can always add more RAM, upgrade the CPU etc. This might be a good solution if you need a computation heavy architecture.

Single-page Application

Throughout the lifetime of web applications, there have been three popular SPA platforms; Java applets, Flash and Javascript. Javascript is the only SPA platform that do not require any special third party applications or plug-ins to run its software, as mentioned earlier in this chapter. SPA adopts the "thin client" idea. In short summary, an SPA transfers its user interface to the client in the browser, and can do most of the computation on the client side which do not require server interaction. By only providing the user interface, an SPA can use it as a template and fill it with content asynchronously without asking the server for entire documents which, as mentioned earlier, will give the user a more responsive interaction with the application, and will relive the web server from computation. The benefit of SPA is that it runs cross-platform as long as the user is using a modern HTML5-supporting browser. This means that users running different operating systems may access the web application without the developer needing to make ports of the software to multiple platforms[25].

Chapter 3

Architecture

3.1 Overall architecture

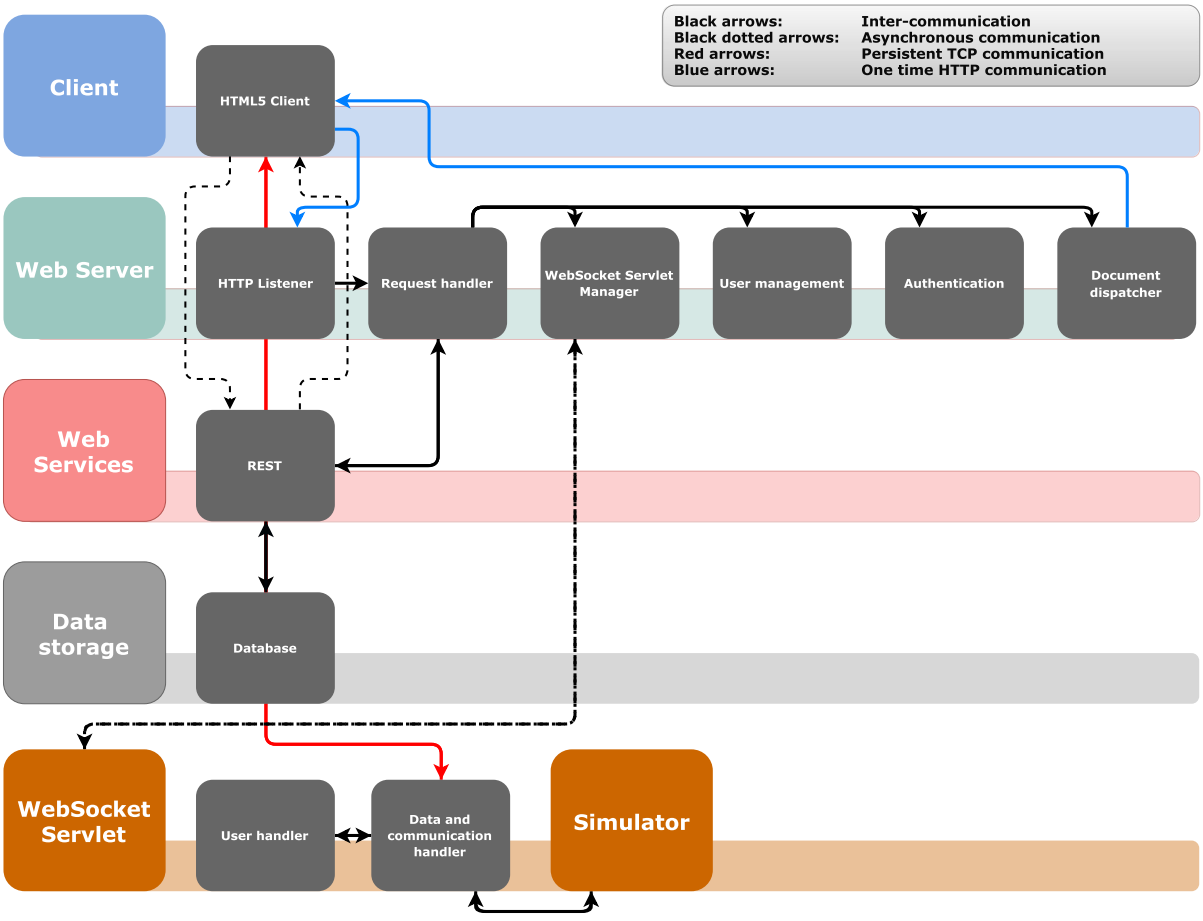

Figure 3.1: Proposed multitiered architecture

Figure 3.1 is an illustration of the multitiered architecture we are proposing. We are proposing a thick client, thin web server and a horizontally scalable thin real-time communication cluster

with WebSockets servlets. Figure 3.1 shows how the different components of the multitiered architecture are communicating, and gives an overview of the core components that are needed to address the problems in the problem statement. From top to bottom, we can see the different tiers, and how they interact with the other tiers. In this chapter, we will go through the different tiers of the architecture in more detail.

3.1.1 Client

The client is a modern browser which has support for HTML5 and has Javascript enabled. The browser will display an SPA to the user. The SPA contains a graphical user interface which have different areas in the interface that displays some type of content.

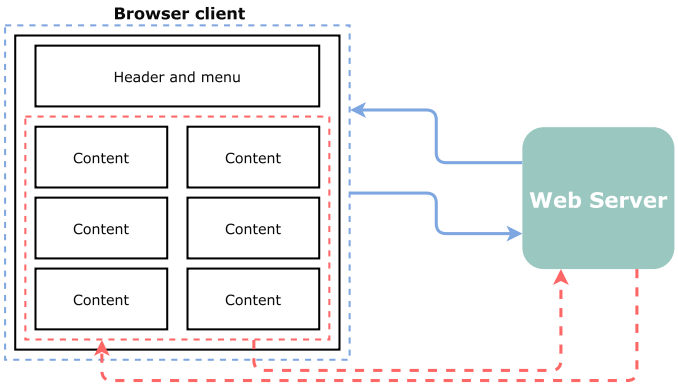

Figure 3.2: Client: Document hand-off and SPA

Figure 3.2 shows an illustration of the SPA. The blue arrows represent HTTP-communication that works in a synchronous fashion. This is the initial communication process the client initiates when contacting the server, where the client requests resources, which in this case is the SPA and its dependencies. It is is a one-time transaction to load the SPA in to the client. All the resources needed for the user to run the SPA initially will be download, but all future interaction with the application will be faster since the user interface is interactable from a local storage. However, the content may be varying. To retrieve new content without polling the server for information or document rendering, we can query the server or the web services for specific parts we want to show or update. This querying happens behind the scenes with Javascript. The asynchronous communication is illustrated with red dotted lines that communicates with the Web services attached to the web server. The asynchronous communication can also go directly to the Web services of the architecture if there is need to retrieve direct information from the database.

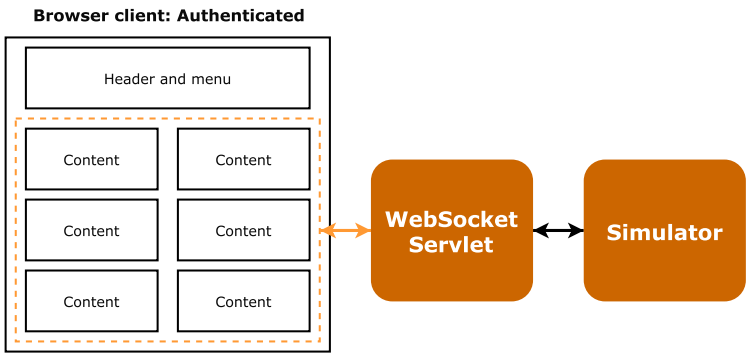

Figure 3.3: Client: Hidden part of the SPA

The SPA also has a hidden service, which only authenticated users may use. This is the part of the SPA that displays the data generated from the simulator in the architecture. This is illustrated with Figure 3.3, where the orange arrows is a bi-directional persistent connection and the black arrows represents inter-communication between the WebSocket servlet and the Simulator, which we will come back to later in this chapter. This is the part of the application where the real-time communication is taking place.

The relationship between client and server is a **thick client** and **thin server**. Some of the logic is being pushed to the client, but only logic that is trivial and won't disturb the service provided by the Web server or the WebSocket servlet.

3.1.2 Web Server

The server is not designed to be CPU-intensive, but rather I/O-intensive to handle more user interaction. Because of the separation of the application from the web server, we do not need to do heavy computation on the server side besides logic verification and request handling. A Web server is a program that uses HTTP to serve files to the requesting clients (browsers). This specific web server is not bound to any type of server software, such as Apache or IIS, making it a general purpose web server. All web servers needs an HTTP-listener and a request handler. This request handler routes the traffic to the parts of the web server where the information requested needs to be handled.

Authentication

There are many ways to authenticate users in web based systems, and we will look at some of the most common:

HTTP Basic authentication, which sends a Base64-encoded concatenated string of your username and password in the header of the HTTP request whenever you try to access a document on the web server which you normally do not have access to. Base64 is a way to encode binary data into a set of characters which is understandable by every modern computer. This is to ensure that no data is lost or modified during transaction. However, this provides a security risk, as HTTP packets are easily intercepted and Base64-encoding is very easily reversible. It also provides some practicality issues, such as the fact there is no practical way of logging out the user without changing its password, nor is there any expiration of the authentication method.

Cookies, which is a local storage of information that is passed between the client and the server. Cookies are usually signed by the server to detect if the cookie has been modified by the client. However, cookies used in authentication are stateful. Meaning that the client and server are both required to keep the authentication record. The cookie that the client is holding is containing a session, which we will come to shortly, and the server needs to keep track of this session, making this authentication method incompatible with the Web services we want to implement.

Sessions are often combined with the use of cookies. The server dispatches a session id to the client which can be incorporated in the cookie. This is an identifier that the server that tracks the users movement. If the session id is incompatible with the area the user wishes to access, they may be rejected. This method is also incompatible with the stateless Web services in the architecture because of the state it brings into the solution. Another aspect of the use of sessions is that the sessions are stored in memory of the server. This strides against the thin server architecture presented and the information is easily lost on server crash, which means the user has to re-initiate the authentication process.

Tokens works like the other methods we have been discussing, and like sessions, they can be stored within cookies. However, there is no persistent s ession for the token on the server-side, which means it is stateless. Since the HTTP protocol in itself is stateless, we have to attach tokens with each request. For every request we do, we tell the server that we are authenticated to access the requested parts. Token authentication is also what we call claim-based authentication, which means that the token contains relevant information about you as an user. Since the token is (usually) signed by the server and if the signature is validated correctly, the user is granted access. Figure 3.4 illustrates the steps of token authentication and Figure 3.5 illustrates the token verification.

Token authentication is the most viable authentication method for this architecture.

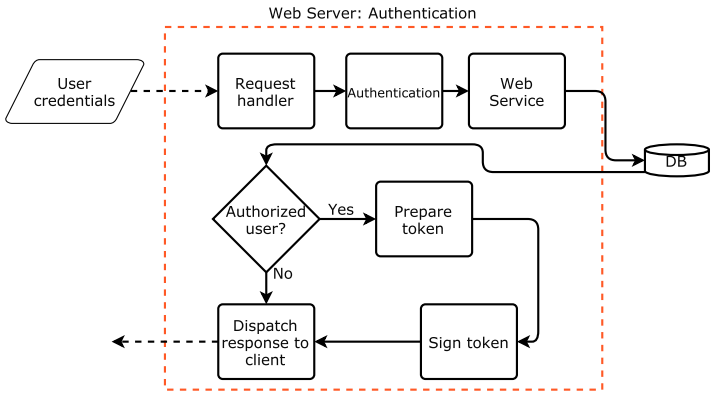

Figure 3.4: Authentication: Token Authentication

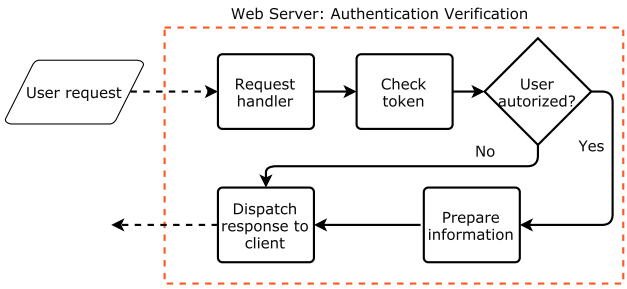

Figure 3.5: Authentication: Token Verification

WebSocket servlet manager

In the presented architecture, the limiting factor is the simulators. As this architecture serves as a general purpose application, the computational limitations provided by the simulators must be thought of. If there are multiple simulators on different machines, the WebSocket servlets needs to be distributed among these simulators. The WebSocket servlet manager is the module of the architecture that keeps track of the information of the distributed servlets.

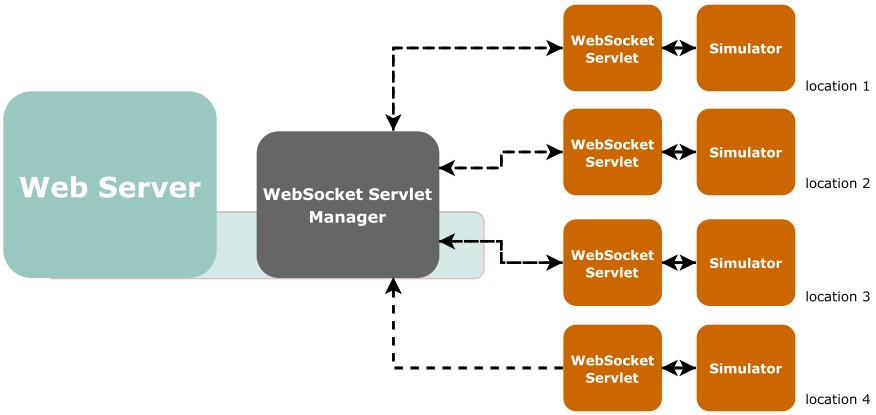

Figure 3.6: WebSocket servlet manager: WebSocket servlet replicas

The WebSocket servlet manager will do periodical polling of the WebSocket servlets or on incoming simulation request, whichever comes first. If there is a servlet available for use, it responds with the servlets IP-address for a client to connect to. If there are no WebSocket servlets available, the WebSocket servlet manager should also have the possibility to instantiate a new servlet-replica.

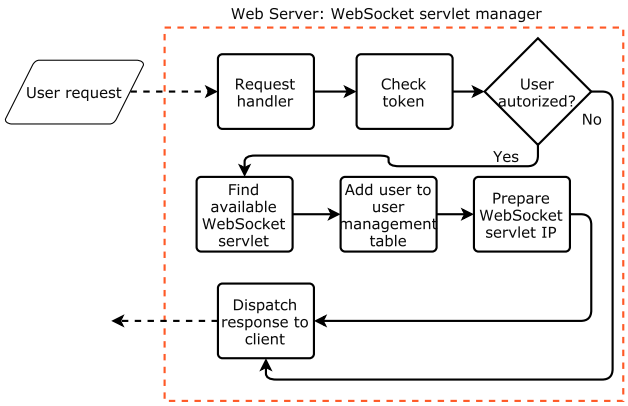

Figure 3.7: WebSocket servlet manager: Find servlet and dispatch information to client

Figure 3.7 illustrates the flow of the user requesting to initiate usage of a simulator. The request handler checks the token to see if the user has the right permissions or is authorized to use the service. If not, the WebSocket servlet manager will automatically decline the request. If the user is authorized, the WebSocket servlet manager will do a check in its table to see which servlet has capacity to take in a new user. The user will then be added to the table of active users and is then assigned an IP to connect. This information will then be dispatched to the client. We take better look at this in the next subsection.

User management

As the simulators is expected to be used in collaboration, it is important to keep track of the active clients interacting with the real-time simulation. When they are connected directly to the WebSocket servlets, they are bypassing the web server when sending and retrieving information. Since the clients are operating with tokens, we do not have any information about the clients state. By keeping track of the users active, we can share this information with the servlets, so they also have an overview over the client situation. If users engages in collaboration, the users is assigned to a **Hub**, that acts as a shared communication channel. An example of a user management table is illustrated with Table 3.1.

User ID	Username	User IP	Hub	Servlet IP
1	Johndoe	71.31.194.213	NULL	10.0.0.5
2	Janedoe	39.239.88.58	1	10.0.0.6
3	Testuser	66.93.68.3	2	10.0.0.7
4	Jacksmith	142.66.247.122	1	10.0.0.6
5	Eliseturner	107.166.234.26	NULL	10.0.0.5

Table 3.1: Example user management table

3.1.3 Web services

The Web services is a module of the Web server in the multitiered architecture. Since the architecture is a thin server architecture, we will propose a lightweight Web services with REST

as discussed in chapter two. However, the REST Web service is not directly exposed in terms of interacting with it directly through an URI. The server request handler that does the routing is the key piece to which parts of the API is to be exposed. This is illustrated in Figure 3.8.

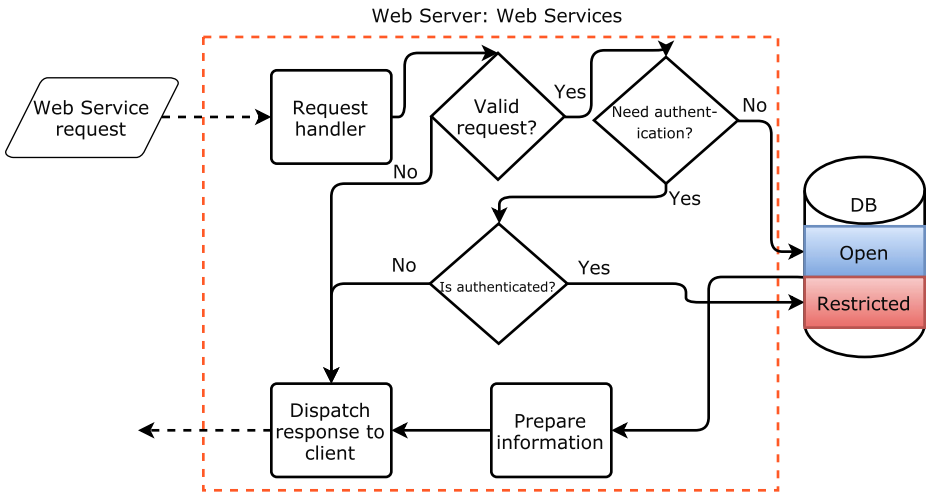

Figure 3.8: WebServices: request flow-chart

Figure 3.8 shows the incoming request from the client to the web server. The request handler takes the request and checks if the request is valid. Here, there are two checks; if the request requires certain permissions or if the request is of a trivial manner. If the request requires an authenticated user, the web service will retrieve the information restricted to authenticated users. If the user is not, it will retrieve the information which is requested that does not require permissions. The level of permissions is defined by the token attached with the request from the client. When the information has been retrieved, the data will be passed to the part of the Web services that encapsulates the information into a specified format or encoding, which can be either JSON, XML or other formats that the developer seems fit for the overall information correspondence in the implementation. We prefer JSON due to the lowered overhead and readability.

3.1.4 WebSocket servlet

The WebSocket servlet consists of small web server. The web server is just meant for dealing with the communication upgrade handshake as described in chapter two. The WebSocket servlet consists of a user handler and a data and communication handler. As mentioned in the earlier sub chapter, the WebSocket servlet manager in the Web Server keeps track of all the users connected to the different WebSocket servlet replicas. This table is shared with each WebSocket servlet, and contains information about the client it is communicating with. If a client drops, the user handler will send a message to the WebSocket servlet manager with the updated information. If a new client connects, the WebSocket servlet will poll the WebSocket servlet manager for an updated table to see if the client exists in the table for the given servlet. The communication handler is the part of the servlet that keeps track over the communication channel that has been established between each user.

The servlet is attached to a given simulator through inter-communication. The architecture do not take in account the computation time of the simulator, only the data transportation. How to instantiate the simulator with the WebSocket servlet to each user or hub, is up to the developer implementing this architecture. This depends on the platform the simulator is running.

Data Delivery

To ensure that the data delivery is ensured in this architecture, after establishing a bi-directional persistent connection, the data delivery handling must consist of an internal handshake. After the handshake is completed, the server will start sending data to the client. The client and server both keep track of the revision of the data that is being sent. When the client receives the next message from the simulator, it will respond with a receipt of the received message to the WebSocket servlet. This ensures that the simulator knows which message to send next. This is illustrated in Figure 3.9. The black stitched arrows represents HTTP-communication, the black solid arrow represents inter-communication as in system messages and the blue arrows represents persistent bi-directional communication.

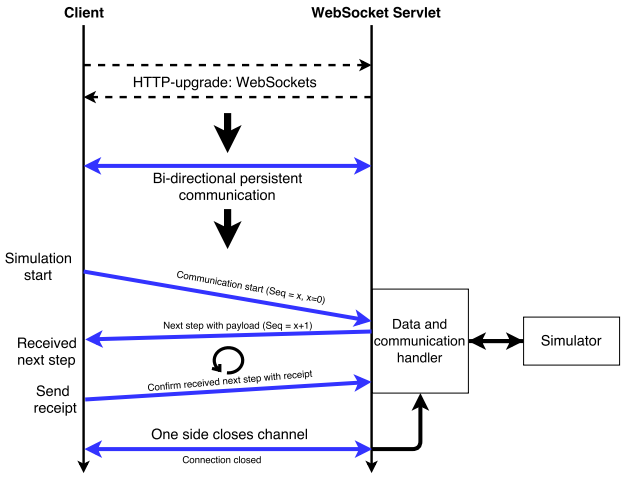

Figure 3.9: Data delivery: Communication process

3.2 Design Alternatives

A design alternative to the proposed architecture is to make the client thin and the web server thick. By doing so, it becomes easier to maintain and control the code that is being dispatched. Because the logic will be on the server side. This means we have to scale the architecture vertically. Instead of focusing on I/O, we will focus on computation. We believe that this will increase the performance in terms of concurrency, but the limiting factor will be the simulators due to not knowing how many instances they are able to support.

Chapter 4

Implementation

The focus of the implementation has relied on the development of the Single-page Application, Web services, authentication and the WebSocket communication service with a simple user management. The SPA, the WebSocket servlet and its communication is the core study. The WebSocket servlets and the simulator are running alongside the web server and not in a distributed fashion as proposed in chapter three. However, the WebSocket servlet and the Web server do share resources, though events that are taking place are happening on two different communication protocols.

The solution is found attached to this document. The solution is called RTSys, which is an acronym for Real-time System, and can be found attached to this thesis. All the components of the prototype are written in Javascript, where the client is using AngularJS framework and the server using Node.js framework. In this chapter we will go trough some of the core functionality of the application and elaborate on some parts which we consider to be crucial. We would like to remind the reader again that in this thesis we consider a simulator as a generic term for a content generator.

4.1 Web Server

Having relatively little experience with the Javascript stack, we found help in manuals and tutorials, particualrly *Javascript: The Good Parts*[26], *Javascript Patterns*[27] and *Javascript with promises*[28]. This literature gives good insight in how to program in Javascript and think in an event-driven fashion. When setting up the Node.js server and the Express[29] framework, we followed *Web Development with Node and Express*[30].

4.1.1 Server framework

Express is a server-side framework in Javascript for Node.js which helps organizing and structure how the user interact with the web application.

```
1  var express = require('express');
2  // use the express framework
3  var app = express();
4  var server = require('http').Server(app);
5  ...
6  app.use(express.static(path.join(__dirname, 'public')));
7  app.use(express.static(path.join(__dirname, 'client')));
8  app.use('/test', express.static(path.join(__dirname, 'simulator')));
9  ...
10 // displays the index page
11 app.use(function(req, res) {
12   res.sendFile(path.join(__dirname, 'client', 'index.html'));
13 });
14 ...
15 server.listen(3000);
```

Example 4.1: Express: Code snippets

In the main server-file, the Node.js-server utilizes Express. In Example 4.1, we import the Express framework and initialize it. After, we use the Express-framework to set up static routing for what is available to be displayed for the clients connecting. A directory in the server structure that is not listed in the Express routing, for example */private/*, will be denied access if a client attempts to access it. The client will then be routed to the index page. Express provides an easy learning curve and is fairly easy to implement. The fact that it provides middleware out of the box makes it easier for request- and error handling such as body parsing. For example, body parsing takes care of the body part of an incoming request, and is exposed to *req.body* as an object which makes it easier to handle. The body parsing is used as the HTTP Request Handler of the prototype. When a client connects, we can see from the function in line 10 that the server will serve the client with the index.html file which is the SPA. We will look into that later in this chapter. When all the routing and conditions are set, we let the Web server listen to incoming connections on port 3000, as we can see on line 15. Whenever a client now accesses the public/external IP of the Web server with the port 3000, they will be served the SPA.

4.1.2 WebSocket servlet

We have chosen Socket.IO as a technology to use for the WebSocket communication because of the abstraction and simplicity it provides for both server- and client-side. Not every client has support for WebSockets, due to limitations in either browser software or operating systems supporting the protocol, so implementing a fallback and aggressive reconnect on connection drops is a tedious task. In case of the client not supporting WebSockets or the client has lost connection, Socket.IO has a graceful fallback to either *Adobe Flash Socket*[31], *AJAX Long Polling*, *AJAX Multipart Streaming*, *Forever iframe* or *JSONP Polling*, whichever the client supports. There is no need to specify which fallback methods are to be used, as the library takes care of this. Like Javascript, Socket.IO is event-driven, meaning that you listen to incoming communication on sockets and respond to the messages you receive from the client with *socket.on('message', function(data)){}* and sending messages to the client with *io.emit('message', data){}*, as we can see from Example 4.2.

```
var io = require('socket.io')(server);
...
io.on('connection', function(socket){
    console.log("Got a connection to Socket.IO, client id "+socket.id);
    var client = new Object();
    client.clientID = socket.id;
    client.simID = "";
    client.simIterations = 0;
    client.simSemaphor = 1;
    client.nOfSimulations = -1;
    client.currentSimulation = "";
    client.logFile = "";
    clientList.push(client);

    socket.on('startSimulator', function(msg) {
        var thisClient = getClient(socket.id);
        thisClient.logFile = "logs/"+ new Date().getTime() + "-log.txt";
        fs.writeFile(thisClient.logFile, '', function(err) {if(err)
        throw err;
        console.log("created log file for client " + socket.id);});
        if (msg === true) {

            thisClient.simID = new contentGenerator("test",0,0,2,1500);
            thisClient.nOfSimulations = thisClient.simID.conf3;
            console.log(thisClient.simID.initialize());
            io.to(client.clientID).emit('ack', true);
        }
    });
    socket.on('nextStep', function(slot){
        var thisClient = getClient(socket.id);
        if (thisClient.simIterations == thisClient.nOfSimulations) {
            console.log("Simulation end for "+ socket.id +". Terminating. Dumping
 ↪  files to log.");
            for (j = 0; j<=thisClient.nOfSimulations; j++) {
                var current = thisClient.simID.displayStepList(j);
fs.appendFileSync(thisClient.logFile, JSON.stringify(current));
            }
console.log("done");
            io.sockets.connected[socket.id].disconnect();
        }
        while (thisClient.simSemaphor == 1) {
            thisClient.currentSimulation =
 ↪  thisClient.simID.simulate(parseInt(slot+1));
            io.to(client.clientID).emit('simulation', thisClient.currentSimulation);
            thisClient.simSemaphor = 0;
        }
    });
        socket.on('receipt', function(object) {
        var thisClient = getClient(socket.id);
        thisClient.simID.receipt(object);
        thisClient.simIterations++;
        thisClient.simSemaphor = 1;
        io.to(client.clientID).emit('receiptAck',object.slot);
        //send to client
    });
});
```

Example 4.2: Socket.IO: Serverside: receiving and sending

Line 4 of Example 4.2, shows the initialization of the communication with the client. This signals that a client has connected. On line 16, the servlet is listening on the sockets and waiting for a message with the contents of **startSimulator**. This is the start signal for the servlet to initiate the content generation from the simulator. The client gets appended to a user list attached to the servlet and gets its own log file. On line 24, we instantiate the simulator. When the simulator has been instantiated, we acknowledge the start and the communication begins. When the server sends out the **ack** to the client which we can see on line 27, the client will automatically respond with a **nextStep**-message. This message tells the WebSocket servlet to contact the simulator to send out the next message.

What happens on the client side, we will cover in the Single-page Application subsection.

4.2 Single-page Application

The client side is implemented with the AngularJS[32], which is an open-source SPA framework. During the implementation of AngularJS, we got a lot of help from a manual called *AngularJS: Up and Running*[33]. The SPA we have implemented relies upon the concept of the event handling of hash changes in the URL-bar. For those who have used a browser, has seen the hash symbol (#) in a URL-bar, much like

```
http://www.your-server.com/index.html#Section3
```

The hash symbol and the following text represents a fragment of the web page displayed, which is used to refer to a certain part of the document. Figure 4.1 illustrates how the different sections of the web application are anchored and how the links can direct the view to a specific part of the document *without refreshing the page*.

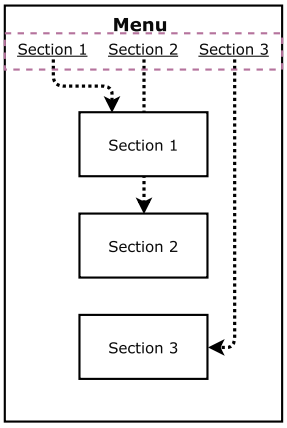

Figure 4.1: SPA: Anchoring

These events can be detected by Javascript with *window.location.hash;*, and can be manipulated with adding an event listener with
window.addEventListener("hashchange", someFunction();. If we can detect changes in the URL without refreshing the document, we can apply behind-the-scenes logic with Javascript. This is what the AngularJS relies upon and delivers a framework for web applications where the view

is separated from the model. When a client accesses the web application, the request will go through the request handler on the Web server and will then route the request to respond with the application, as shown in the subsection about the Server framework.

```
1  <!DOCTYPE html>
2  <html ng-app="RTSys">
3  <head>
4    <title>RTSys</title>
5    <base href="/">
6    <link rel="stylesheet" href="/lib/bootstrap/css/bootstrap.min.css">
7    <link rel="stylesheet" href="/lib/bootstrap/css/bootstrap-theme.min.css">
8  </head>
9  <body ng-view>
10
11
12   <script src="lib/angular/angular.min.js"></script>
13   <script src="lib/angular/angular-route.min.js"></script>
14   <script src="webserver.min.js"></script>
15
16 </body>
17 </html>
```

Example 4.3: SPA: Bootstrapping

As we can see from Example 4.1, the client will be given the *index.html* document as shown in Example 4.3. The document is only providing the template of the application and the dependencies which are needed to run the application to the client. Line 6 and 7 shows the Bootstrap library[34], which is a front-end framework that enables us to build a graphical user interface. The dependencies that are being loaded is the graphical user interface libraries, the AngularJS dependencies and routes for the SPA. These dependencies are only fetched once throughout the interaction with the web application.

We have mentioned routing on the Web server in Example 4.1. Since the SPA is separated from the Web server, we do not ask the Web server for content, so we have to do some internal routing to know which content to bring up from the Web server. All this is still being done behind the scenes, as intended with the SPA model. This is done by the main-file which is located in the root folder of the solution.

```
1  (function () {
2
3    angular.module('RTSys', ['ngRoute']);
4
5    // angular routing
6    function config ($routeProvider, $locationProvider) {
7      $routeProvider
8        .when('/', {
9          templateUrl: 'home/home.view.html',
10         controller: 'homeCtrl',
11         controllerAs: 'vm'
12       })
13         .when('/simulator', {
14           templateUrl: 'simulator/simulator.view.html',
15           controller: 'simulatorCtrl',
16           controllerAs: 'vm'
17         })
18 ...
19       .otherwise({redirectTo: '/'});
20
21     $locationProvider.html5Mode(true);
22   }
```

Example 4.4: SPA: Routing

Example 4.4 shows the internal routing of the SPA. When the client requests a page, this is the module that will handle it. If the path is not defined, the client will automatically redirected to the front page, as we can see in line 19. As we can see from line 9 and line 14, when the client locates to one of the paths, either "/" or "/simulator", it will request a view for the different paths which are defined by the "templateUrl" parameter. These views are masked by the paths and are not displayed for the client. Instead of showing

`http://www.your-page.com/app_client/simulator/simulator.view.html`

the client URL will simply show

`http://www.your-page.com/simulator`

As we mentioned earlier, all these actions relies upon the "hashchange" event listener, but in the examples above, #-symbols are not visible. If the clients browser has HTML5 support, which is one of the criterias in the proposed architecture, we can mask them with using HTML5-mode. $locationProvider, as seen Example 4.4 line 21, is the provider which stores the applications linking paths. With the HTML5-mode activated, the #-symbol will disappear, making the SPA act like a traditional website, but load the content dynamically, since everything happening after "http://www.your-page.com/" are events that can be monitored and manipulated.

Authentication

A web application needs to have an authentication handler. Rather than developing a basic one, we find that Passport.js is a middleware that can be implemented in the Express framework. Passport is an extensible library that has the capabilities to authenticate users using single sign-ons from social networking sites such as Twitter, Facebook or LinkedIn which provides

an OAuth[35]. OAuth stands for Open Authentication, and is a concept that relies on authentication towards accepted third-party applications to grant access to your service. The different authentication mechanisms that Passport provides are known as *strategies*. These strategies are not included in the primary library, but rather modules that can be implemented after need. The Passport solution in this implementation is using a standard strategy called local-strategy. This strategy relies only on the users verifying their login credentials with that is stored in the database. The user uses the web application to either register or log in with the server, which sends the user information to the REST Web service for authentication. When the user is authenticated, Passport will dispatch a signed JSON Web Token (JWT) to the client in the HTTP response header, which we can see in Example 4.5, where the client later uses on the service to prove authenticity towards the server.

```
var crypto = require('crypto');
var mongoose = require( 'mongoose' );
var jwt = require('jsonwebtoken');
...
userSchema.methods.setPassword = function(password){
  this.salt = crypto.randomBytes(16).toString('hex');
  this.hash = crypto.pbkdf2Sync(password, this.salt, 1000, 64).toString('hex');
};

userSchema.methods.validPassword = function(password) {
  var hash = crypto.pbkdf2Sync(password, this.salt, 1000, 64).toString('hex');
  return this.hash === hash;
};


userSchema.methods.generateJwt = function() {
  var expiry = new Date();
  expiry.setDate(expiry.getDate() + 7);

  return jwt.sign({
    _id: this._id,
    email: this.email,
    name: this.name,
    exp: parseInt(expiry.getTime() / 1000),
  }, "masterthesis");
};
```

Example 4.5: Authentication: Token signing

When the token has been signed, it is dispatched to the client so it can be saved in the local-storage for further interaction with the web application. This is being handled by an authentication service. This module takes care of all logic that involves the authentication on the client side, from logging in, registering users, saving token to client, getting token from client, posting authentication details to the Web service, to removing the token (logging client out). Example 4.6 shows a snippet of the processing.

```
1  (function () {
2    angular
3      .module('RTSys')
4      .service('authentication', authentication);
5    authentication.$inject = ['$http', '$window'];
6    function authentication ($http, $window) {
7      var saveToken = function (token) {
8        $window.localStorage['auth-token'] = token;
9      };
10     var getToken = function () {
11       return $window.localStorage['auth-token'];
12     };
13 ...
14     register = function(user) {
15       return $http.post('/api/register', user).success(function(data){
16         saveToken(data.token);
17       });
18     };
19     login = function(user) {
20       return $http.post('/api/login', user).success(function(data) {
21         saveToken(data.token);
22       });
23     };
24     logout = function() {
25       $window.localStorage.removeItem('auth-token');
26     };
```

Example 4.6: Token signing

We can see from line 15 and 20, that the requests are being forwarded with the POST-verb to the Web services.

The provided token is a Base64-encoded string which consists of a header, claims and a signature. The header tells us what kind of type the token is, the claim gives information that the token claims it is, and the signature is a verifying HMAC SHA256 hash function to detect if there has been used a different secret for signing the token[36]. The encoded JWT looks like the one illustrated in Example 4.7. When we unravel the encoding, we get the following head, claim and signature in Examples 4.8, 4.9 and 4.10

```
eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJfaWQiOiI1NzM5ZjRmNGU4YzViNmJjMDdh
NWRmYjYiLCJlbWFpbCI6InRlc3RAdGVzdC50ZXN0IiwibmFtZSI6InRlc3QiLCJleHAiOjE0N
jQ3MjgzMDgsImlhdCI6MTQ2NDEyMzUwOH0.OT46-5aGo4bHNP03F3jS-65Uny2uz7G4F19uFZ
Onjic
```

Example 4.7: Encoded JSON Web Token

```
{
  "typ": "JWT",
  "alg": "HS256"
}
```

Example 4.8: JWT Header: Token type and Algorithm

```
{
  "_id": "5739f4f4e8c5b6bc07a5dfb6",
  "email": "test@test.test",
  "name": "test",
  "exp": 1464728308,
  "iat": 1464123508
}
```

Example 4.9: JWT Claim: Payload consisting of user information

```
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  masterthesis
)
```

Example 4.10: JWT Signature: Verification with secret key: "masterthesis"

Remark: we can extend the authorization tokens to clients to define which access rights they possess. This can be an extension or a complement to the Web services, where we define which paths are restricted or open directly in the token claims.

Web services

The layout for the REST webservices we have implemented is located in the /api/ directory. In the /routes/ folder we find the routes that the Web service is using to dispatch and control the flow of information. From the authentication section, we do HTTP posts to both /api/register and /api/login, and in Example 4.11 is where we handle them. Here we can provide different URI's we utilize to perform actions through the Web services. We include controllers that are passed along with the HTTP verbs in the Web service. These controllers contain the schema information filters the requests.

```
1  var ctrlProfile = require('../controllers/profile');
2  var ctrlAuth = require('../controllers/authentication');
3  // profile
4  router.get('/profile', auth, ctrlProfile.profileRead);
5  // authentication
6  router.post('/register', ctrlAuth.register);
7  router.post('/login', ctrlAuth.login);
```

Example 4.11: Web Services

On line 4 in Example 4.11, we take in two arguments, which is a request and response. How this is processed is shown in Example 4.12, which we will come back to shortly. We see that in order to request the profile information, we need to pass in the auth, which is the authorization token, and we want to respond with the profileRead-function response.

```
1  var mongoose = require('mongoose');
2  var User = mongoose.model('User');
3  module.exports.profileRead = function(req, res) {
4    if (!req.payload._id) {
5      res.status(401).json({
6        "message" : "Unauthorized"
7      });
8    } else {
9      User
10       .findById(req.payload._id)
11       .exec(function(err, user) {
12         res.status(200).json(user);
13       });
14   }
15 };
```

Example 4.12: Web Services: Requestion profile information

In Example 4.12, we load the model for the database entry we want to look up, and then check if the authorization token is valid as we can see in line 4. We respond either with a Unautorized message or with a success-response, which are both being delivered to the client in a JSON-encoded response.

This Web service can be extended further by adding the different HTTP verbs to the routing, and from what we can see of Figure 3.8 of our proposed architecture, we can separate the requests from being open or restricted.

WebSocket servlet communication

As mentioned earlier in this chapter, we will continue with the client side for simulation. This code is located in the */sim/* directory.

```
1  <script src="socket.io-1.4.5.js"></script>
2  <script src="jquery-1.12.3.min.js"></script>
3  ...
4  <input type="button" id="Start" value="Start">
5
6  <script>
7      var socket = io.connect("http://<external ip>:3000/");
8      var currentSlot = -1;
9      var timeSlot = 0;
10     $('#Start').click(function(){
11         socket.emit('startSimulator', true);
12         console.log("start sim");
13     });
14     socket.on('ack', function(response) {
15         socket.emit('nextStep',currentSlot);
16         console.log("ack the conn. next step in simulation");
17     });
18     socket.on('receiptAck', function(response) {
19         socket.emit('nextStep', currentSlot);
20     });
21     socket.on('dump', function(data) {
22         var output = JSON.stringify(data);
23         var input = $("#serverDump").val() + output + "\n";
24         $("#serverDump").val(input);
25     });
26     socket.on('simulation', function(data) {
27         var output = JSON.parse(data);
28         currentSlot = output.slot;
29         var input = $("#simulationOutput").val() + output.slot + ":" + output.value +
    ↪  "\n";
30         $("#simulationOutput").val(input);
31         timeSlot = new Date().getTime();
32         var receipt = JSON.stringify({'slot':currentSlot,'receiptTime':timeSlot});
33         try {
34             socket.emit('receipt', receipt);
35         } catch(e) {
36             console.log(e);
37         }
38     });
39 </script>
```

Example 4.13: SPA: WebSocket servlet communication

Example 4.13 shows snippets of the code for the WebSocket communication. For the communication to start, we have to load necessary dependencies. Socket.IO requires a client-library and relies on the jQuery-library. These are imported in line 1 and 2. The rest of the view are two boxes in which the information from the server will be located, so the user can see the live updates from the simulator. On line 7, we connect to the WebSocket servlet with the IP. This will initiate an HTTP Upgrade-process and result in a bi-directional communication link between the client and the server. When we click the button that is on line 4, we send the **startSimulator** command to the server through the new connection over WebSockets. When the client receives the next sequence from the WebSocket servlet, it will respond with a receipt, which is generated on line 35. This will go on until either the client or server drops or closes the connection, as we can see from Figure 3.9 in chapter three.

Chapter 5

Test and Results

In this chapter, we will explain how we deploy and start the implemented solution, show how it runs and do some stress-testing of the solutions. We present some data that has been generated from the stress-testing and some graphs to illustrate the runs and finally we will discuss the findings of the data and graphs.

5.1 Simple User Manual

This is a simple user manual to set up the implemented solution

First, you need to acquire a machine or use a virtual machine for the solution and have a modern browser such as Google Chrome, Firefox or Safari.

1. **Download and install Ubuntu Server**

Visit `http://www.ubuntu.com/download/server` to download Ubuntu Server. Install it to your machine or your virtual machine. In our setup, we have used Ubuntu Server 14.10 LTS[37] with the Linux 3.16 kernel.

2. **Install Node.js via package manager**

Install Node.js to the server with the following command in terminal:

```
curl -sL https://deb.nodesource.com/setup_4.x | sudo -E bash -
sudo apt-get install -y nodejs
```

3. **Unpack the implemented solution**

Unpack the implemented solution from the Appendix to a folder seemed fit. We have used */rtsys/* as a folder for our solution.

4. **Check which dependencies are needed**

Locate the selected directory and open package.json. This file contains 13 dependencies needed to run the solution and which versions of the dependencies are required, which includes; Express, JSON Web Token, Passport, Socket.IO, Mongoose and other middleware. All of these can be installed separately. Alternatively, you can locate to the folder in terminal and run:

```
npm install
```

This starts the Node Packet Manager, and will automatically read the *package.json*-file and install the necessary dependencies. Note: some of the packages requires root user, so it might be necessary to start the command with *sudo*.

5. **Set a port for the web server**

In the solutions folder, *webserver.js* is the application that Node.js is going to run. Open it and edit last line which says *server.listen();*, and add the port you want to use. This port is shared with Socket.IO.

6. **Port-forward the port you want to use**

Most networks have limited their in- and outgoing ports for security reasons. Enter the router or modem that is being used and assign in and outgoing ports for the IP the machine or the virtual machine is running to the port you specified in the previous step. If you are unable to do so, contact your network administrator.

7. **Run the solution**

The solution should now be properly set up. To start the server, run this command in the terminal:

```
node webserver.js
```

Once it is up and running, you can access the application using your browser. You can either navigate to your internal IP or your external IP with the port you have chosen in step 5 in this manner:

```
http://your.ip.here:port
```

5.2 Sample run

Client side

When the server has started, one can navigate to the SPA at (*http://your.ip.here:port*), and the index page will load, as seen in Figure 5.1.

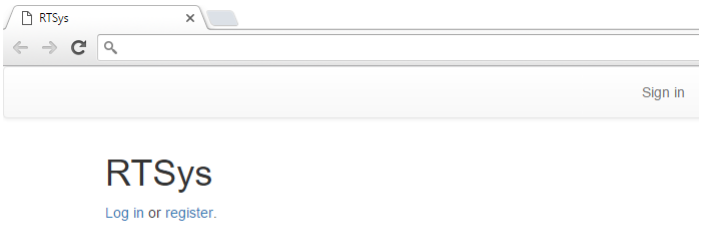

Figure 5.1: Client: Index page

From here you can either log in or register if you do not have an account with the service. If you choose to register, the register page will be displayed as seen in Figure 5.2. If if you have provided valid credentials in the register form, you will be automatically logged in and directed to the profile page as seen in Figure 5.4. If you are already registered with the service, you simply need to provide the credentials and log in through the log in page as seen in Figure 5.3. As on the register page, if they are authenticated, they will be directed to the profile page. Once the client is authenticated, you can start a simulation form the simulation page as seen in Figure 5.5.

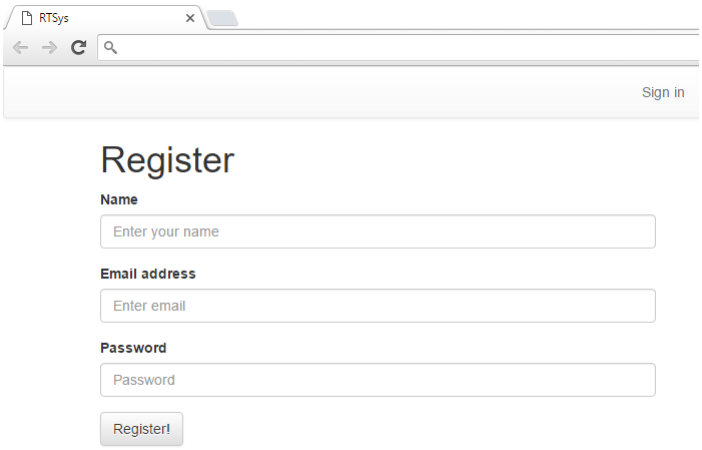

Figure 5.2: Client: Register page

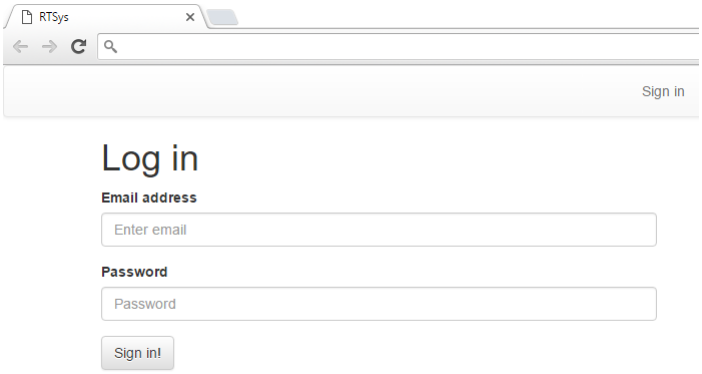

Figure 5.3: Client: Login page

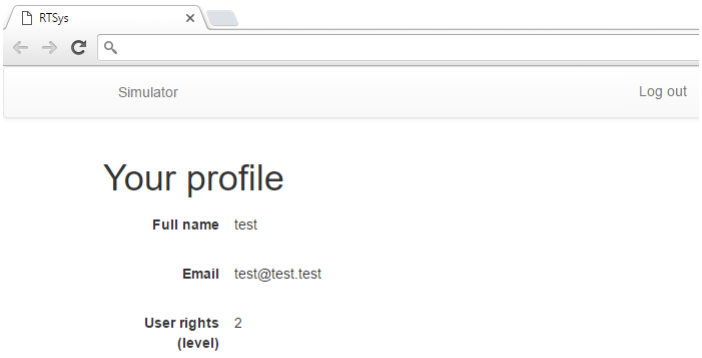

Figure 5.4: Client: Profile page

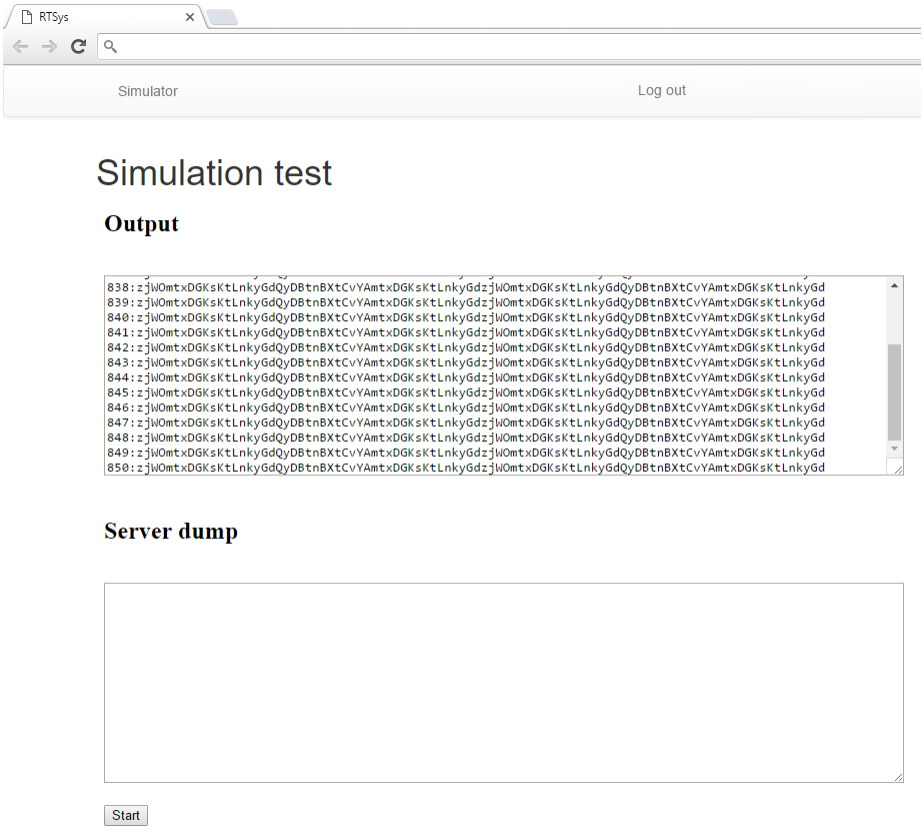

Figure 5.5: Client: Simulation page

When you reach the simulation page, the clients sends an HTTP upgrade header to the server to prepare for the WebSocket transmission. When the user clicks the *Start* button, the client and the server starts an internal handshake and the results generated by the simulator is transferred over to the Output window as seen in Figure 5.5. For the debugging purposes, at the end of the data transfer, the server will dump the data that was collected during transmission; *sequence numbers, data, sending time, return time* and *round trip time*.

Server side

Once the Web server has been initiated, clients are now able to connect. The server will display a log of connected clients, and which users that have started a simulation. It will also display where during the simulation a given user is, as seen in Figure 5.7. If there are many clients connected to the server at a given time running a simulation, the debug output may be heavily delayed, because of the limitations in the console.

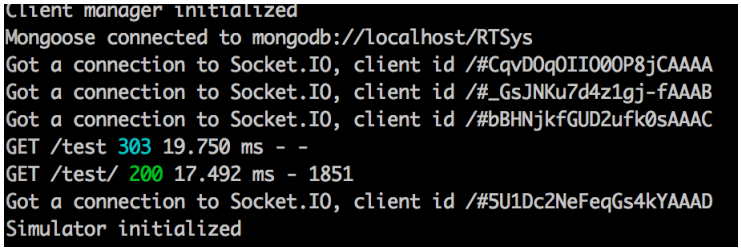

Figure 5.6: Server: Starting the server and a simulation

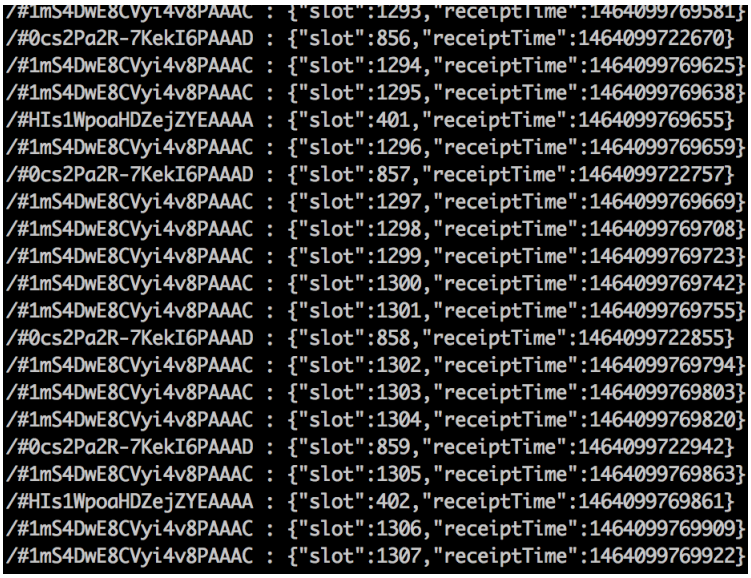

Figure 5.7: Server: Three clients running simulation simultanously with debug print

5.3 Testing

We are interested in seeing how the WebSocket servlet handles user concurrency and load. The goal of the stress-testing if to see if the quality of service of the WebSocket servlets deteriorates. To do that, we emulate real users interacting with the service and log the activity. The data that is collected is from logging each clients session with the simulator, measuring the RTT. This client-server communication is referred to in this chapter as a **sequence**. The RTT of these sequences are measured from the server side to avoid the issue of clock synchronization.

We have made a script that generates virtual browser clients that accesses the implemented solution. Each virtual browser client starts the communication process with the WebSocket

servlet and sends a signal to the WebSocket servlet to instantiate a simulation instance. This way, we can emulate multiple real users interacting with the service. The virtual clients are on the Unix-servers that are provided by the University of Stavanger. The servers used for this testing are *pitter[21-37].ux.uis.no, gorina2.ux.uis.no* and *gorina3.ux.uis.no*.

5.3.1 Virtual clients

The script we made is based on Python, PhantomJS[38] and Selenium[39]. PhantomJS is a headless browser based on WebKit[40] which only exists as an instance or script on the running machine. Selenium automates the process of testing websites, by accessing the DOM of the page and interacting with it without the use of a real user. By combining PhantomJS and Selenium, we create n number of instances of a web client to connect to the server. When they are all instantiated, the script will then execute the simulation process as shown in Figure 5.5 for each instance.

What does it mean using a headless browser and which advantages does it provide?
A headless browser is a browser which does not provide a GUI. It still will interpret the DOM and Javascript that is in the solution, and will work as any other web browser. They controlled via an API or a console, which makes it easy to automate the process. The code shown in 5.1 creates *20* instances of virtual clients and start them sequentially within a short period of time. They will run simultaneously a few seconds apart. In contrast to what is being shown to a real user, the virtual clients do not receive a data dump from the server, but the server logs and saves the data which contains the *slot numbers, data, sending time, return time* and *round trip time*. This is only running server-side since we have not implemented any clock synchronization between the client and the server.

```
import string
import urlparse
import urllib
import sys

from selenium import webdriver
from selenium.webdriver.support.ui import Select
from selenium.webdriver.support.ui import WebDriverWait
from selenium.common.exceptions import NoSuchElementException

class VirtualClient(object):
    def __init__(self,instance):
        self.url = "http://<external-ip>:3000/test"
        self.driver = webdriver.PhantomJS()
        self.driver.set_window_size(1120, 550)
        self.instance = instance;
        print "Creating Virtual Client #",instance + 1

    def startSimulation(self):
        self.driver.get(self.url)
        try:
            self.driver.find_element_by_id('Start').click();
            print "Starting Simulation for Virtual Client #", self.instance + 1
        except NoSuchElementException:
            print "No Start-button found"
if __name__ == '__main__':
    myList = []
    for i in range(0,20):
        myList.append(VirtualClient(i))
    for j in range(0,20):
        myList[j].startSimulation()
```

Example 5.1: Virtual Clients: Code

5.3.2 Results

The logs that have been generated are of server stress-testing with the virtual clients. The virtual client script has generated different scenarios during stress-testing, as listed in Table 5.1.


```
{"slot":0,"message":"zjWOmtxDGKsKtLnkyGdQyDBtnBXtCvYAmtxDGKsKtLnkyGdzjW
OmtxDGKsKtLnkyGdQyDBtnBXtCvYAmtxDGKsKtLnkyGd","sendTime":1464190318391,
"returnTime":1464190318396,"RTT":5}{"slot":1,"message":"zjWOmtxDGKsKtLn
kyGdQyDBtnBXtCvYAmtxDGKsKtLnkyGdzjWOmtxDGKsKtLnkyGdQyDBtnBXtCvYAmtxDGKs
KtLnkyGd","sendTime":1464190318400,"returnTime":1464190318404,"RTT":4}
{"slot":2,"message":"zjWOmtxDGKsKtLnkyGdQyDBtnBXtCvYAmtxDGKsKtLnkyGdzjW
OmtxDGKsKtLnkyGdQyDBtnBXtCvYAmtxDGKsKtLnkyGd","sendTime":1464190318407,
"returnTime":1464190318410,"RTT":3}
```


Example 5.2: Example log content generated locally with 3 transactions

The content of the log files is JSON-encoded as shown in Example 5.2. However, the RTT-

times in the example are relatively low (~4 ms) due to the client and the server communicating on the same LAN. The example is only illustrative. In Table 5.1, we have collected the mean RTT for the service with different scenarios, where we combine messaging with low, middle and high load with few, medium and many clients. First, we will do a general analysis to see how the service performs in terms of RTT and concurrency. Later in this chapter, we will look more in depth of the statistical data provided by the testing. The collected results are from a stress-test with virtual clients.

Clients	**# Sequences**	**Mean RTT (ms)**	**# Clients dropped**
20	1500	51	0
20	5000	112	0
20	10000	154	0
50	1500	114	0
50	5000	110	0
50	10000	200	0
100	1500	160	2
100	5000	721	1
100	10000	2032	1
150	1500	356	1
150	5000	1099	1
150	10000	2212	9

Table 5.1: Results from the virtual client stress-testing

At 20 clients and 50 clients, we have the most stable performance. The RTT is below the threshold of 250ms that we described in the introduction, and we experience no client drops. The communication of the sequences is still considered real-time.

At 100 clients, the RTT for 1500 messages is acceptable. However, if we increase the sequence load for each client, we see that the delay quickly increases, and goes over the defined threshold.

With 150 clients, we have a service that is not delivering acceptable performance. We also experience an increase in client drops. At 150 concurrent users, the WebSocket servlet is utilizing between 90.6 % and 99.7 % of the CPU as shown in Figure 5.8. This is the absolute most clients we can connect to the implemented WebSocket servlet.

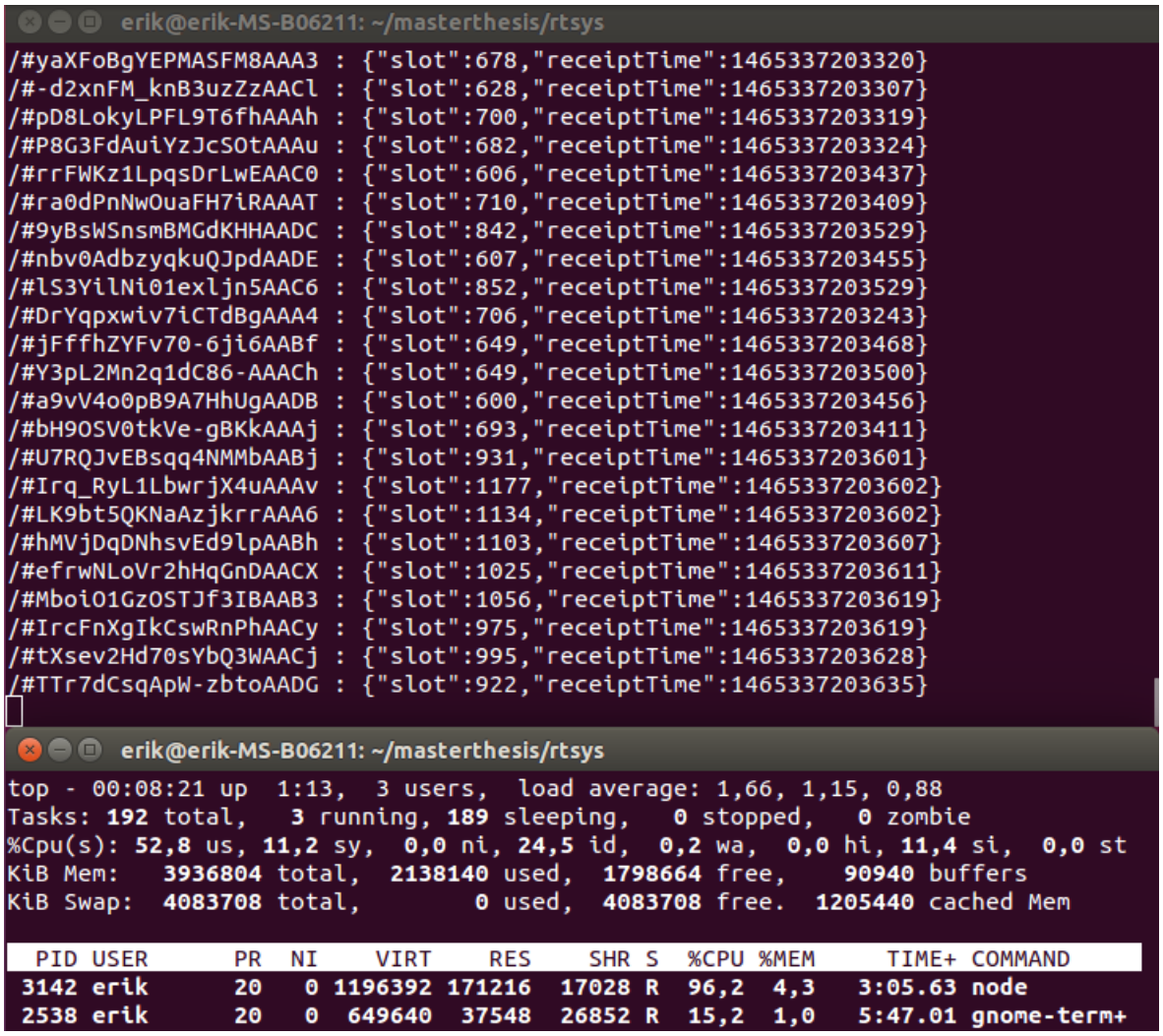

Figure 5.8: Web server and WebSocket servlet: CPU load

# Clients	# Sequences	RTT High	RTT Low	Q1	Q3	IQR
20	1500	79	27	40.0	65.0	25.0
50	1500	1250	21	72.0	152.0	80.0
100	1500	2170	23	86.0	250.25	164.25
150	1500	5490	22	211.75	487.25	275.5
20	5000	2981	21	95.0	131.0	36.0
50	5000	210	27	67.0	155.0	88.0
100	5000	5487	21	430.75	1050.25	619.5
150	5000	3869	22	600.75	1638.0	1037.25
20	10000	10870	22	116.0	190.0	74.0
50	10000	399	26	112.0	288.0	176.0
100	10000	66600	22	993.0	3033.25	2040.25
150	10000	74540	22	1114.75	3327.25	2212.5

Table 5.2: Stress-testing: Values in milliseconds

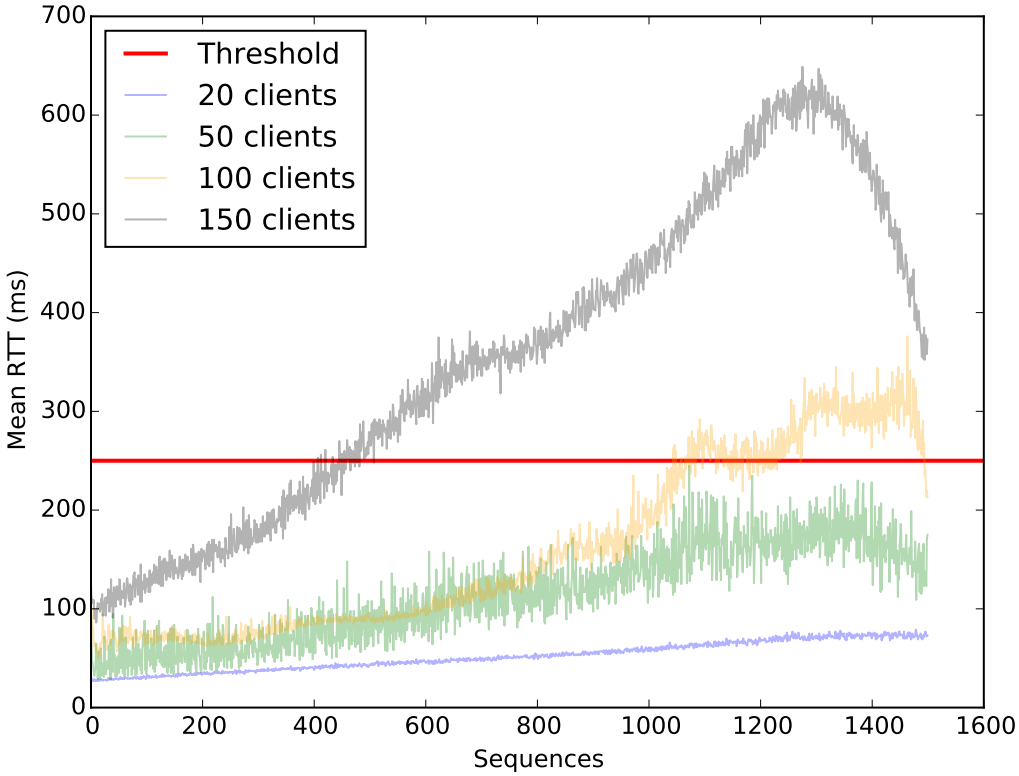

Figure 5.9: Stress-testing: Plot of 1500 simulations

5000 sequences

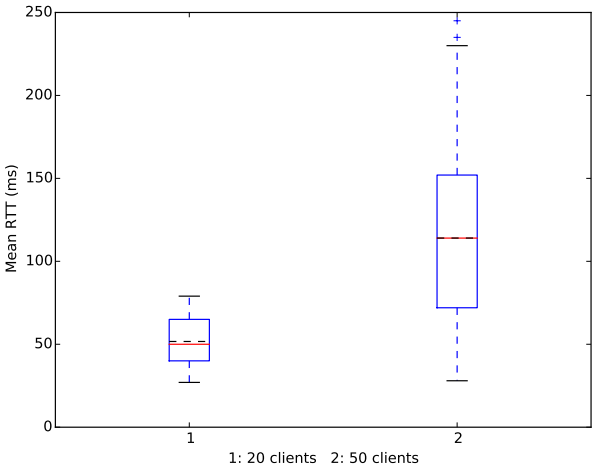

Figure 5.10: Stress-testing: Box plot of 1500 sequence part 1

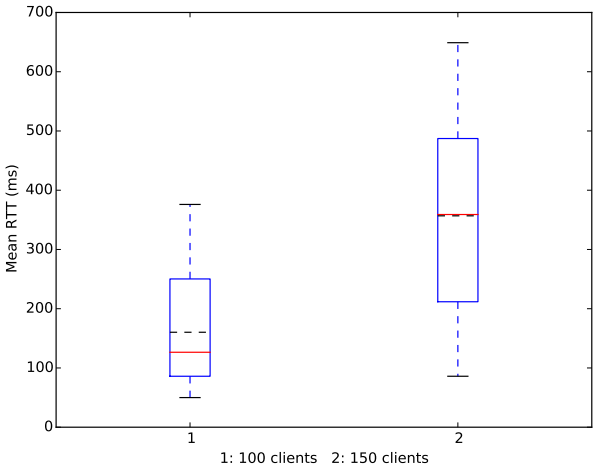

Figure 5.11: Stress-testing: Box plot of 1500 sequences part 2

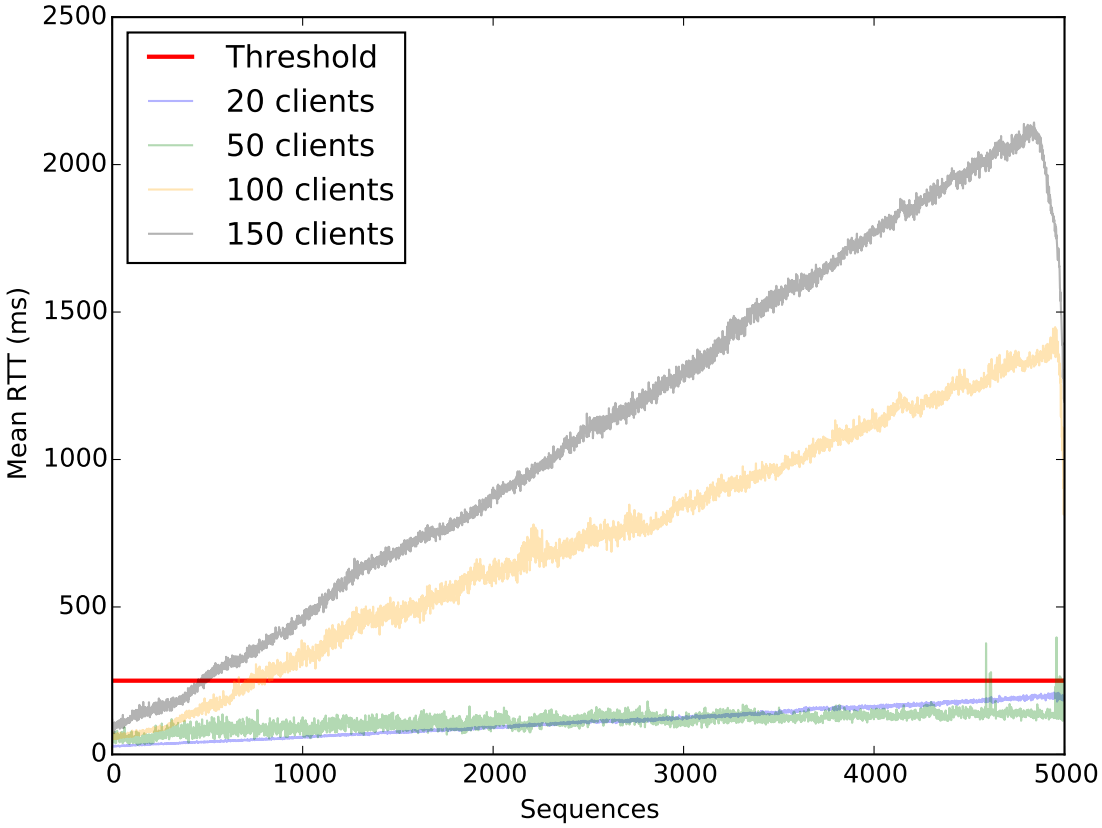

Figure 5.12: Stress-testing: Plot of 5000 simulations

5000 sequences

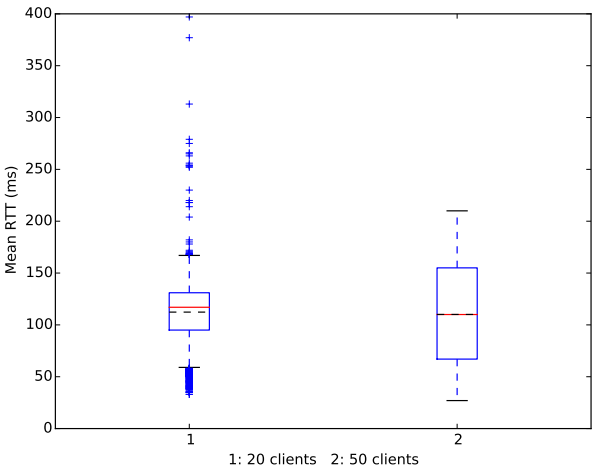

Figure 5.13: Stress-testing: Box plot of 5000 sequence part 1

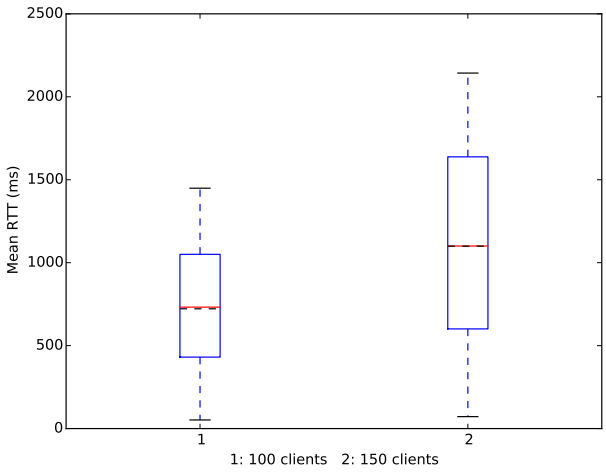

Figure 5.14: Stress-testing: Box plot of 5000 sequences part 2

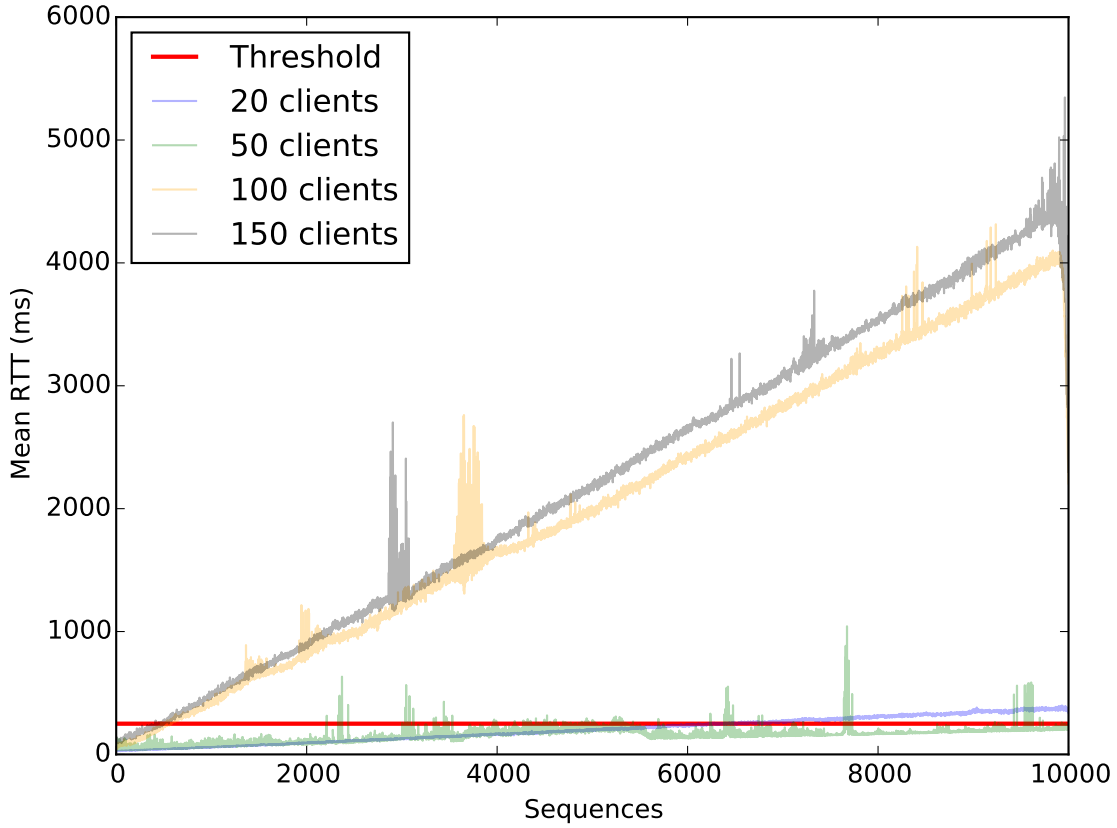

Figure 5.15: Stress-testing: Plot of 10000 simulations

10000 sequences

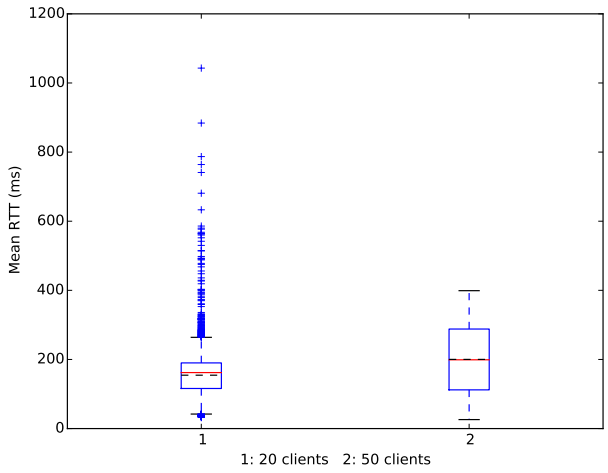

Figure 5.16: Stress-testing: Box plot of 10000 sequences part 1

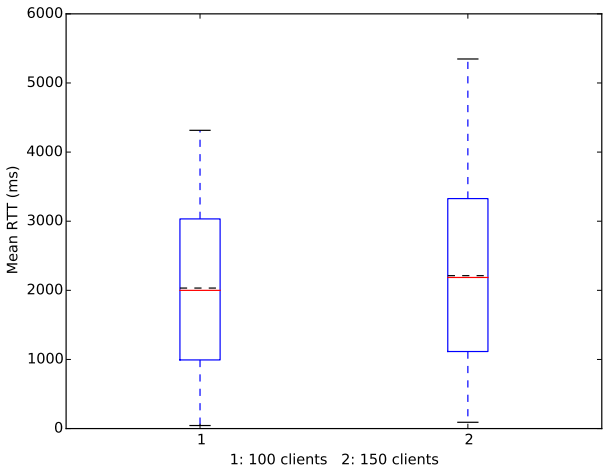

Figure 5.17: Stress-testing: Box plot of 10000 sequences part 2

5.3.3 Results discussion

Before we carried out the testing, we had an expectation of the graph reaching an equilibrium point where the mean RTT would be relatively constant to clients and load, but we see that the RTT increases over time when we increase the number of clients above 50.

Producing a function of the plots or making a conclusion around the findings would be meaningless, due to so many factors that can affect the result. These factors can be, but are not limited to: processing delay, queueing delay, transmission delay and propagation delay. These factors themselves give a complex profile of the latency the service can experience. In addition to these, we have noise, jitter, network interference, segmented TCP packets, encoding / decoding data, geographical distance or hardware, in addition to many more.

We are under the assumption that these results appear due to the event-driven WebSocket servlet, which are putting all the messages in a queue. This quickly congests the event-loop on the server side which gives increased latency. We cannot draw any definite conclusions of whether the results are affected by the factors listed above, or if there is a bug in the implementation. What we can say about the results is that the current implementation shows that it is saturated to ~50 clients with low, medium and high load. These findings are empirical and what we can do with this, we will discuss this in the conclusion in chapter six.

Chapter 6

Discussion

In this chapter, we will discuss our work. We will look at the originality of the work, what limitations were present, what we have learned, what our conclusion is in regard to the problem statements in chapter one and what is future work.

6.1 Originality of this work

We have not found any resources on the Internet or in literature that resembles the proposed architecture, but we are certain that the proposed architecture is not unique. WebSockets and REST are technologies that have been extensively researched and tested for quite some time. The prototype that is produced, is a common approach to how Single-page Application and REST Web services are designed, and its structure its widely known and documented. However, we have not found solutions that are exactly like the set up of the real-time transportation, nor the benchmarking. There are many tools that are designed to benchmark Node.js and Socket.IO, but not in the form of emulating real users, which we have done.

6.2 Limitations

Almost every student project we have some kind of limitations, and this project is no different. There have been on hardware, development platforms, network, network capture and programming languages. In the following subsections we briefly discuss the aforementioned limitations.

6.2.1 Hardware

The machine we used for the testing is a *MSI Wind Box DC110*, with a *Intel Celeron 847 Dual Core Processor* and *2GB DDR3 RAM*. Dealing with a lot of requests from the client during test periods, and inexperience with Javascript, lead to the machine often running out of memory (server side). There were also some significant delays in accessing the server through SSH, using version-control software Git[41] and starting scripts that builds your solution such as Gulp[42]. Building the solution when developing could vary between 5 and 20 seconds, and the server startup could take up to 15 seconds. This lead to very much waiting in times where only small changes were being done.

6.2.2 Development platforms

The first revision of the architecture was deployed on a Windows Server 2012 R2, where the WebSocket framework used for testing out the architecture was SignalR (C#) on IIS. Midway in the implementation, the server that we were granted started shutting down unexpectedly during initial runs. The unexpected shutdowns was investigated, but the shutdown logs showed no conclusive answers and extensive troubleshooting of the problem did not resolve the issue. This resulted in starting over with adapting the code to a different server technology and a new WebSocket library, respectively Node.js on Ubuntu Server with Socket.IO.

6.2.3 Network

Testing on the local network gives significantly lower response time compared to latency outside the LAN. The network is also limited by the provider, and some network providers limits the speed to applications that are being exposed to the Internet. We also got reports from the Internet Provider that they had noticed activity resembling a DoS-attack, which lead to throttling of the connection.

6.2.4 Network capture

During the stress testing of the implementation, we tried to log and analyze the data that were captured during the runs with Wireshark. However, the output files produced from Wireshark for each run was hard to analyze because of the fact that the WebSocket payload were under certain circumstances spread across multiple WebSocket frames, making it hard to patch together the correct ones.

6.2.5 Programming language

When working with programming language one is not familiar with, and have a goal of completing a task, it is easy to take a few short cuts. When the original architecture was to be implemented in C#, we did not realize that single-thread event-driven programming with Javascript would be so much different. This lead to many unexpected shutdowns of the Node.js server due to unintentional blocking, locks or infinite loops. Dealing with event-driven programming language, you have to think differently compared to Java or C#.

6.3 Learning experience

This thesis has been a great learning experience for us. Although we have previously touched some areas within web programming, such as PHP and HTML, in addition to having a course in web programming at the university. By implementing a prototype with a hands-on approach, we gained better insight in how the different aspects and components of the implementation actually interact.

The web is a wast field with many areas of research. We have to take in consideration everything from hardware to software, from network models and topologies to architectures.

For implementation, communication protocols that are used by web solutions should be understood as well.

The areas we have deeper understanding of is the usability and implementation of REST Web services, implementing Single-page Applications, socket programming (WebSockets) and handling user concurrency with an asynchronous approach instead of a threaded/process approach. The exploration of Single-page Application was a particularly interesting experience. To emulate traditional web directory browsing with displaying content asynchronously is a very powerful tool, and should not be undermined. The exploration of WebSocket was somewhat a tedious task. We had issues with the service halting, which often is hard to find the root cause of.

To be able to create a prototype in the short amount of time given, is a milestone which we are happy to achieve.

6.4 Conclusion

We are successful in producing a prototype to address the problems we have defined in our problem statement, which we will now go through.

1. **What steps are necessary to make a modern web application interactive?**

Modern interactive web applications are called Single-page Applications. We set the basic technology context for modern web applications in chapter two, laid down the building blocks in the third chapter, and finished with an implementation demonstration in chapter four. Making a Single-page Applications is constructed by using Javascript to asynchronously fetch content behind the scene(s), while masking the hash change in the URL-bar and update the view without refreshing the document.

This requires a modern web browser.

2. **How will the proposed architecture for this project scale?**

In chapter three we explained the architecture, where we looked into the different aspects of the proposed architecture. By having a thick client, and a thin server, we can distribute the computation load to the clients instead of doing the heavy lifting on the server side. The server side is still doing work like performing trivial logical operations, such as verification and request handling. The WebSocket servlets are proposed to be dispatched in a distributed manner. This concludes with the client being horizontally scalable, as well as the WebSocket servlets. However, the WebSocket servlets scaling is bound to their performance.

3. **Are WebSockets or WebSocket based libraries mature enough to handle reliable transportation of data?**

During the stress-testing in chapter five, we experienced client drops, but we did not experience packet drops or information loss. This was expected since TCP natively has a method of retransmitting lost or damaged packets. We did not experience that Socket.IO changed its transportation method, so the WebSocket connection was kept stable during the testing. Given that the actions are performed in the environment where HTML5 and WebSockets are activated.

We can observe, but can not conclude that this is a universal truth for all WebSocket libraries and we can not put any claim to the statement when the load increases. WebSockets may be the reason we experience the results we see.

4. **Will multiple persistent connections result in a deterioration of the quality of service the system will provide?**

We cannot conclude that it is universally accepted that such a service only can handle a maximum of 50 concurrent users as we can see from the results in chapter five, since the prototype implementation is set up in a specific way, and another implementation of the architecture can be set up fundamentally different.

In this case, we suggest two approaches that can address this problem:

1. **Find the root cause.** We can study the implementation and its setup and look for bugs and try to optimize or fix it from there.

2. **React symptomatically.** We can study a problem from a replication point of view. With a feasible saturation point and given that the system operates without faults, we can use the client saturation value as a pointer to dynamically scale the WebSocket servlets with instantiating new replicas on demand.

We should assume that the system performance is non-deterministic. It is hard to predict performance improvements or deterioration. The minimum requirement for such a system performance is to observe and react.

6.5 Future work

The architecture presented is far from complete as it is described and the prototype implementation only covers a small area of the overall picture. There is a lot of work remaining and further areas we can research.

1. Re-implement the architecture with other hardware specifications, server operating system and implementation frameworks to see how other solutions differ from the one presented.
2. Design an architecture benchmark tool for this purpose.
3. Design a Web service for WebSockets, where the message exchange is descriptive. This can be used to further develop an API for the general purpose architecture.
4. Implement the WebSocket servlet manager / load balancer to dynamically instantiate new replicas.
5. Apply the architecture to a real-life scenario, instead of a dummy-simulator used for prototyping.
6. Extend the interaction of the application to include collaboration / sharing simulator instances.

Bibliography

[1] Kristian Johannessen. Real time web applications. Master's thesis, University of Oslo, 2014. https://www.duo.uio.no/handle/10852/42049 Online; accessed February 2nd, 2016.

[2] Lightstreamer Srl. Lightstreamer. https://www.lightstreamer.com/. Online; accessed April 12th 2016.

[3] Microsoft. Asp.net signalr. http://www.signalr.net/. Online; accessed January 28th, 2016.

[4] Open-Source. Socket.io. http://www.socket.io/. Online; accessed January 28th, 2016.

[5] Øyvind Raasholm Tangen. Real-time web with websocket. Master's thesis, University of Oslo, 2015. https://www.duo.uio.no/handle/10852/44808 Online; accessed February 7th, 2016.

[6] Cisco. Open systems interconnection (osi) reference model. http://www.cisco.com/cpress/cc/td/cpress/fund/ith/ith01gb.htm. Figure 1-2. Online; accessed May 26th 2016.

[7] Jakob Nielsen. Response times: The 3 important limits. *Evidence-Based User Experience Research, Training, and Consulting*, January 1st, 1993. Online; accessed Febraury 22nd, 2016.

[8] W3C Working Draft 30 January 2014. Xmlhttprequest level 1. https://www.w3.org/TR/XMLHttpRequest/. Online; accessed May 27th, 2016.

[9] Dave Roos. How activex for animation works. http://entertainment.howstuffworks.com/activex-for-animation1.htm. Online; accessed May 27th, 2016.

[10] Bruce Jay Birrell, Andrew D., Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems (TOCS)*, February 1984. Online; accessed May 25th, 2016.

[11] 2003 Microsoft, March 28. How rpc works. https://technet.microsoft.com/en-us/library/cc738291%28v=ws.10%29.aspx. Online; accessed May 26th, 2016.

[12] G. Mein, S. Pal, G. Dhondu, T.K. Anand, A. Stojanovic, M. Al-Ghosein, and P.M. Oeuvray. Simple object access protocol, September 24 2002. US Patent 6,457,066.

[13] R. Khalaf W. Nagy N. Mukhi S. Weerawarana F. Curbera, M. Duftler. Unraveling the web services web: an introduction to soap, wsdl, and uddi. *IEEE Internet Computing*, 6(2):86–93, 2002. Online; accessed March 25th, 2016.

[14] W3C Note 15 March 2001. Web services description language (wsdl) 1.1. `https://www.w3.org/TR/wsdl`. Online; accessed May 27th, 2016.

[15] Roy Thomas Fielding. *Architectural styles and the design of network-based software architectures*. PhD thesis, University of California, Irvine, 2000.

[16] R. Fielding, J. Reschke. Hypertext transfer protocol (http/1.1): Semantics and content. *Internet Engineering Task Force (IETF)*, June 2014. Online; accessed March 25th, 2016.

[17] Savas, Robinson Ian Webber, Jim, Parastatidis. *REST in Practice*. O'Reilly, first edition, 2010.

[18] W3C Member Submission 31 August 2009. Web application description language. `https://www.w3.org/Submission/2009/SUBM-wadl-20090831/`. Online; accessed May 27th, 2016.

[19] A.,Isode Ltd. Fette, I., Google Inc,Melnikov. Implementing remote procedure calls. *Request for Comments: 6455*, December 2011. Online; accessed May 26th, 2016.

[20] Internet Engineering Task Force HTTP Working Group. Http/2 frequently asked questions. `https://http2.github.io/faq`. Online; accessed May 26th, 2016.

[21] Oracle. Sun java system communications services 6 2004q2 enterprise deployment planning guide. `https://docs.oracle.com/cd/E19522-01/817-6096/architecture.html`. Online; accessed May 29th, 2016.

[22] Oracle. Oracle toplink, developer's guide. january 2006. `http://docs.oracle.com/cd/B25221_04/web.1013/b13593/undtldev011.htm`. Online; accessed May 29th, 2016.

[23] Oracle. Oracle toplink, developer's guide. january 2006. `http://docs.oracle.com/cd/B25221_04/web.1013/b13593/undtldev010.htm`. Online; accessed May 29th, 2016.

[24] Oracle. Database concepts. chapter 10: Application architecture. `https://docs.oracle.com/cd/B19306_01/server.102/b14220/dist_pro.htm`. Online; accessed May 29th, 2016.

[25] Michael S Mikowski and Josh C Powell. Single page web applications. *B and W*, 2013.

[26] Douglas Crockford. *JavaScript: The Good Parts*. O'Reilly Media, 1st edition, 2008. ISBN: 9780596158736.

[27] Stefanov Stoyan. *JavaScript Patterns*. O'Reilly Media, 1st edition, 2010. ISBN: 9780596158736.

[28] Daniel Parker. *JavaScript with Promises*. O'Reilly Media, 1st edition, 2015. ISBN: 9781449373207.

[29] Node.js Foundation. Express framework. http://expressjs.com/. Online; accessed April 14th 2016.

[30] Ethan Brown. *Web Development with Node and Express*. O'Reilly Media, 1st edition, 2014. ISBN: 9781491949283.

[31] Adobe. Socket - as3. http://help.adobe.com/en_US/FlashPlatform/reference/actionscript/3/flash/net/Socket.html. Online; accessed June 2nd, 2016.

[32] Google. Angularjs framework. https://angularjs.org/. Online; accessed April 14th 2016.

[33] Brad Seshadri, Shyam, Green. *AngularJS: Up and Running*. O'Reilly Media, 1st edition, 2014. ISBN: 9781491901892.

[34] Bootstrap. Bootstrap. http://getbootstrap.com/. Online; accessed February 5th, 2016.

[35] PassportJS. Documentation. https://www.passportjs.org/docs.

[36] Atlassian. Understanding jwt. https://developer.atlassian.com/static/connect/docs/latest/concepts/understanding-jwt.html. Online; accessed April 26th, 2016.

[37] Canonical Ltd. Ubuntu server 15.10. http://www.ubuntu.com/server. Online; accessed February 9th, 2016.

[38] Canonical Ltd. Phantomjs. http://phantomjs.org/. Online; accessed May 19th, 2016.

[39] Ariya Hidayat. Selenium. http://www.seleniumhq.org/. Online; accessed May 19th, 2016.

[40] Apple Inc. Webkit. https://www.webkit.org/.

[41] Linus Torvalds. Git. https://www.git-scm.com/.

[42] Gulp. Gulp. http://www.gulpjs.com/. Online; accessed February 6th, 2016.

Appendix A

Source code

The source code can be found attached to this PDF in a compressed format. This file contains the solution, the virtual client script, the result parsing scripts and the graphs produced from the parsing scripts.

The raw log files are too large in size to include in this report, so they can be found at `http://www.esolhaug.no/master`

The directories which contain the log files are structured as such:
[number of clients]c[**number of sequences**]s