

## **REALTIDSKOMMUNIKATION I FULL DUPLEX ÖVER WEBBEN**

En prestandajämförelse mellan olika  
webbteknologier

## **REAL-TIME FULL DUPLEX COMMUNICATION OVER THE WEB**

A performance comparison between  
different web technologies

Examensarbete inom huvudområdet datalogi  
Grundnivå 30 Högskolepoäng  
Vårtermin 2012

Elof Bigestans

Handledare: Henrik Gustavsson  
Examinator: Mikael Berndtsson

## **Abstract**

As the web browser becomes an increasingly powerful tool for the average web user, with more features and capabilities being developed constantly, the necessity to determine which features perform better than others in the same area becomes more important. This thesis investigates the performance of three separate technologies used to achieve full-duplex real time communication over the web: short polling using Ajax, server-sent events and the WebSocket protocol. An experiment was conducted measuring the performance over three custom-built web applications (one per technology being tested), comparing latency and number of HTTP requests over 100 messages being sent through the application. Additionally, the latency measurements were made over three separate network conditions. The experiment results suggest the WebSocket protocol outperforms both short polling using Ajax and server-sent events by large margins, varying slightly depending on network conditions.

**Keywords:** Real-time communication, websocket, server-sent events, short polling, Ajax, performance

# Table of Contents

<b>1</b>	<b>INTRODUCTION.....</b>	<b>2</b>
<b>2</b>	<b>BACKGROUND.....</b>	<b>3</b>
2.1	REPEATED POLLING.....	3
2.2	CONTINUOUS CONNECTION .....	5
2.3	WEB SERVERS.....	7
<b>3</b>	<b>PROBLEM FORMULATION .....</b>	<b>8</b>
3.1	DELIMITATIONS .....	8
<b>4</b>	<b>METHOD .....</b>	<b>10</b>
4.1	EXPERIMENT SCOPE .....	10
4.2	INDEPENDENT VARIABLE: THE EXPERIMENT APPLICATIONS.....	10
4.3	DEPENDENT VARIABLES: THE PERFORMANCE ATTRIBUTES.....	11
4.4	PERFORMING THE MEASUREMENTS.....	11
4.5	NETWORK CONSIDERATIONS .....	12
4.6	ALTERNATIVE METHODOLOGIES .....	12
4.7	OTHER CONSIDERATIONS .....	13
<b>5</b>	<b>IMPLEMENTATION.....</b>	<b>14</b>
5.1	LITERATURE REVIEW .....	14
5.2	PILOT STUDY.....	15
5.3	BUILDING THE EXPERIMENT APPLICATIONS.....	17
5.4	PERFORMING THE MEASUREMENTS.....	29
<b>6</b>	<b>RESULTS AND ANALYSIS .....</b>	<b>32</b>
6.1	LOCAL MACHINE LATENCY MEASUREMENTS .....	32
6.2	LAN LATENCY MEASUREMENTS.....	33
6.3	4G LATENCY MEASUREMENTS.....	35
6.4	LATENCY ANALYSIS .....	37
6.5	NUMBER OF HTTP CONNECTIONS .....	38
<b>7</b>	<b>CONCLUSION.....</b>	<b>40</b>
7.1	SUMMARY.....	40
7.2	DISCUSSION.....	41
7.3	FUTURE WORK.....	42
	<b>REFERENCES .....</b>	<b>44</b>
	<b>APPENDIX A – PILOT APPLICATION SOURCE CODE.....</b>	<b>47</b>
	<b>APPENDIX B (PILOT STUDY MEASUREMENT DATA) .....</b>	<b>51</b>
	<b>APPENDIX C – WEBSOCKET APPLICATION SOURCE CODE.....</b>	<b>52</b>
	<b>APPENDIX D – SSE CLIENT APPLICATION SOURCE CODE .....</b>	<b>57</b>
	<b>APPENDIX E – AJAX APPLICATION SOURCE CODE.....</b>	<b>62</b>
	<b>APPENDIX F – MEASUREMENT SCRIPT .....</b>	<b>68</b>
	<b>APPENDIX G – LOCAL MACHINE MEASUREMENT DATA.....</b>	<b>70</b>
	<b>APPENDIX H – LAN MEASUREMENT DATA .....</b>	<b>72</b>
	<b>APPENDIX I – 4G MEASUREMENT DATA .....</b>	<b>74</b>

# 1 Introduction

The web is becoming an increasingly ubiquitous tool for average computer users. There are web applications springing up capable of handling everything from image processing, text messaging to video conferencing (Qurashi & Anwar, 2012). One aspect of web application development that has been notoriously difficult using traditional HTTP technologies is real-time full duplex communication between client and server (Agarwal, 2012). In this context, real-time duplex communication refers to bidirectional communication between client and server without requiring full page refresh within the browser.

Traditionally, 3<sup>rd</sup> party plugins and applications would have to be used and embedded on web pages to achieve real-time duplex communication (Alvestrand, 2012). This would also mean that the user would have to download and install these third party tools. However, with the advent of HTML5 and other innovative web technologies implemented directly in the browser, a number of new technologies have become available that allow full duplex real-time communication without the use of 3<sup>rd</sup> party plugins. The purpose of this thesis is to compare these technologies.

At the moment, there are several technologies available to web developers to achieve real-time full duplex communication (Alvestrand, 2012). Among them is, for example, repeated short polling (often using Ajax), long polling, WebSockets and HTML5 Server-Sent Events. These technologies have different capabilities, limitations, availability (in terms of browser support) and performance. As such, it might be a daunting task for a web developer to decide on which of these technologies to use. This study aims to make that easier by making an evaluation on one of these aspects, namely the performance of the web technologies.

A pilot study was conducted and a minimal pilot application was constructed to test the feasibility of the chosen methodology. After having determined the validity of the methodology, three applications were constructed to house the experiment. One application for each technology to be tested was created. The applications function as instant text messaging chat rooms where multiple users can connect and communicate.

Experiments were performed on the applications. The experiment consisted of two types of measurements: measuring the latency of the application over 100 messages being sent, and measuring the number of HTTP requests required to send 100 messages. The latency measurements were performed over 3 separate network conditions: over a local machine, with the server and client software on the same computer, over a local area network and over a wireless 4G connection.

## 2 Background

HTTP is a request/response protocol – under typical operation the client establishes a connection with a HTTP server and requests a resource. The server responds with the requested resource and the connection is terminated. In the standard HTTP model, the server cannot initiate a connection to a client nor send an HTTP response that has not first been requested – thus, asynchronous communication between a server and client is not possible (Loreto, Saint-Andre, Salsano, & Wilkins, 2011). In order to get around this limitation of HTTP, a number of technologies and techniques have been popularized in recent years – in this section a few of the most popular techniques will be described.

These technologies can be coarsely categorized in two groups:

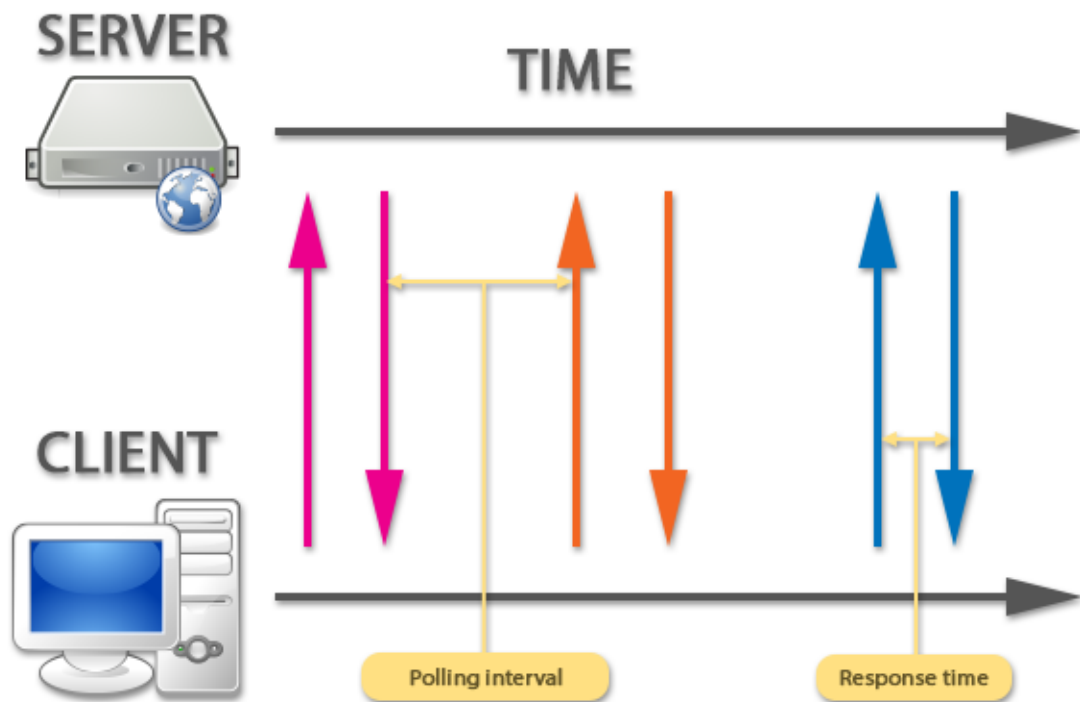
- Repeated polling, constantly opening and closing new connections
- Continuous connection, keeping a single connection open

### 2.1 Repeated polling

Repeated polling essentially means that the client repeatedly sends requests for resources from a web server and compares them to the latest fetch to see if there've been any updates (Pimentel & Nickerson, 2012). If any updates are detected, the data is handled by the client. There is no continuous connection in use in this scenario – rather, entirely new HTTP connections are established and closed with every repetition.

Polling and long polling are two commonly used techniques that utilize repeated polling. They're described later in this chapter.

Repeated polling carries quite a bit of overhead and unnecessary transmissions, as the client needs to keep opening and closing connections to the web server to see if any updates are available. When using this method, the developer needs to determine the polling interval, i.e. the time to wait between each poll.



**Figure 1: Typical short polling operation. Each arrow set represents a separate request and response. There is no continuous connection between client and server, only repeated requests (polls)**

### 2.1.1 Short polling using Ajax

Ajax is not a single protocol or method – rather, it’s a collected set of interrelated web development techniques that allow asynchronous data retrieval and presentation on a web page without a full page refresh. The XMLHttpRequest JavaScript API is used to enable asynchronous data retrieval – this data can then be used to update a web page using HTML and the Document Object Model (Garret, 2005).

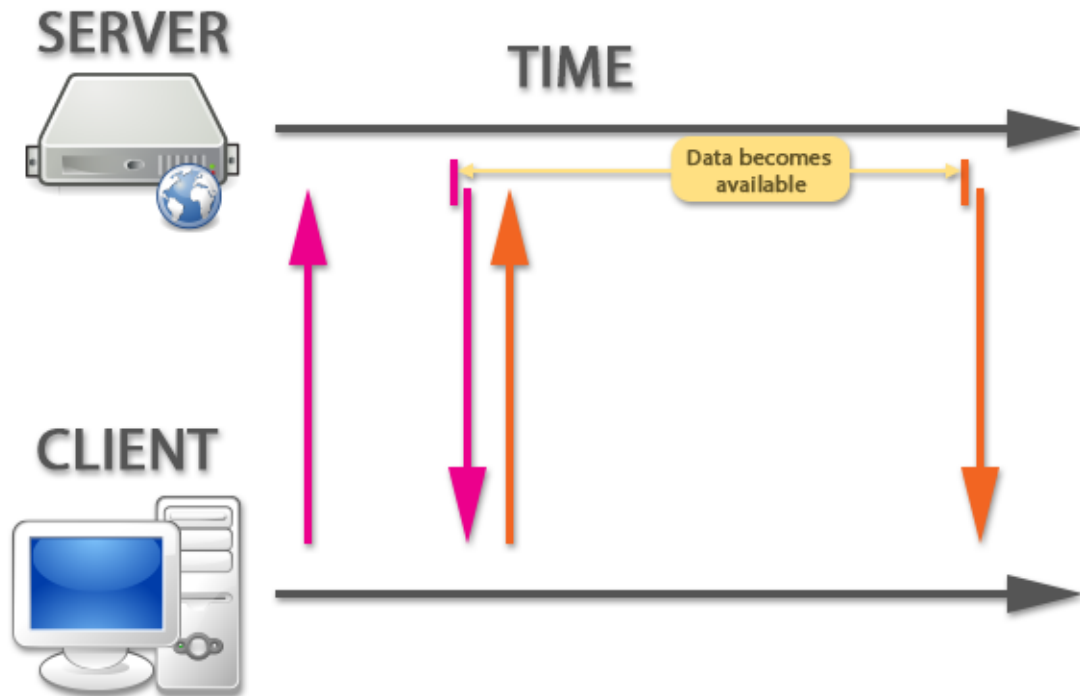
This can then be used to continuously poll a server “in the background” of a web page, i.e. happening without refreshing the page or letting the user know about the continuous flow of HTTP requests being sent. In this way, bidirectional communication can be simulated – the client compares the retrieved data with the existing data and if there’s an update, the web page is updated with the new information (Shuang & Feng, 2013). This technique is called “short polling”, i.e. polling the server repeatedly in short-term intervals to detect updates.

### 2.1.2 Long Polling

Long polling attempts to decrease the unnecessary traffic generated by the repeated polling of a server. When utilizing short polling, a whole new HTTP request/response cycle has to be made each time the client fetches updates, which produces a lot of overhead even when there are no updates to present. If there’s no updated data to send to the client, the server will instead keep the HTTP request open until the data has been updated or until a fixed time-out period is reached. A typical long polling cycle looks like this (Loreto, Saint-Andre, Salsano, & Wilkins, 2011):

1. The client requests a resource and awaits a response

2. The server does not respond immediately, but rather keeps the request open until there are new updates to send or until a fixed timeout is reached
3. When an update exists the server responds to the open request
4. The client typically sends a new request immediately to start a new long poll



**Figure 2: Typical long polling operation. Each arrow set represents a separate request and response. The server holds the request and does not respond until there is new data to present to the client. Once the client receives a response, a new request is immediately dispatched.**

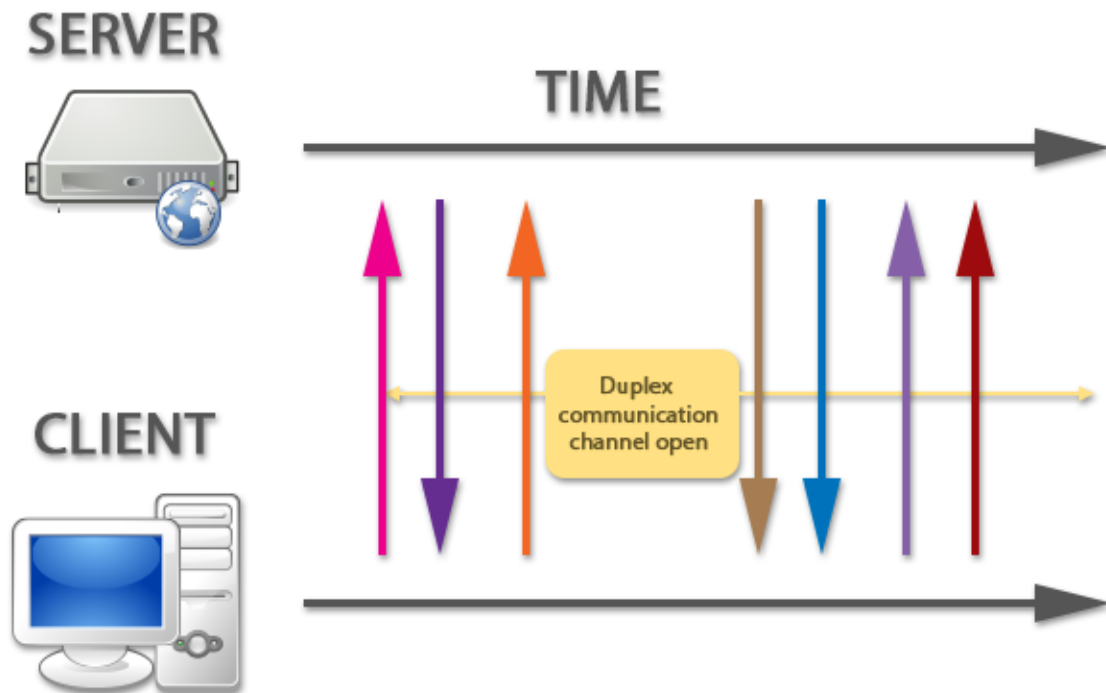
## 2.2 Continuous connection

Continuous connections attempt to circumvent the typical request-response structure entirely, instead replacing it with a continuous connection that is kept open indefinitely (Loreto, Saint-Andre, Salsano, & Wilkins, 2011). The WebSocket protocol and Server-Sent Events are two examples of technologies using continuous connection.

### 2.2.1 The WebSocket protocol

WebSocket is a protocol that enables full duplex communication between a client and a web server. The protocol works over TCP (just like HTTP) and was created as an alternative to repeated polling (Fette & Melnikov, 2011). An interactive communication session is established between the client and the web server – while open, both the server and the client can send messages to each other without relying on a request-response structure. See figure 3 for an illustration of how WebSocket works.

Because WebSocket works over a single, continuous TCP connection it can be utilized to efficiently and effectively process the flow of data between client and server with a minimal amount of overhead and while providing a high level of scalability (Zhao, Xia, & Le, 2013).



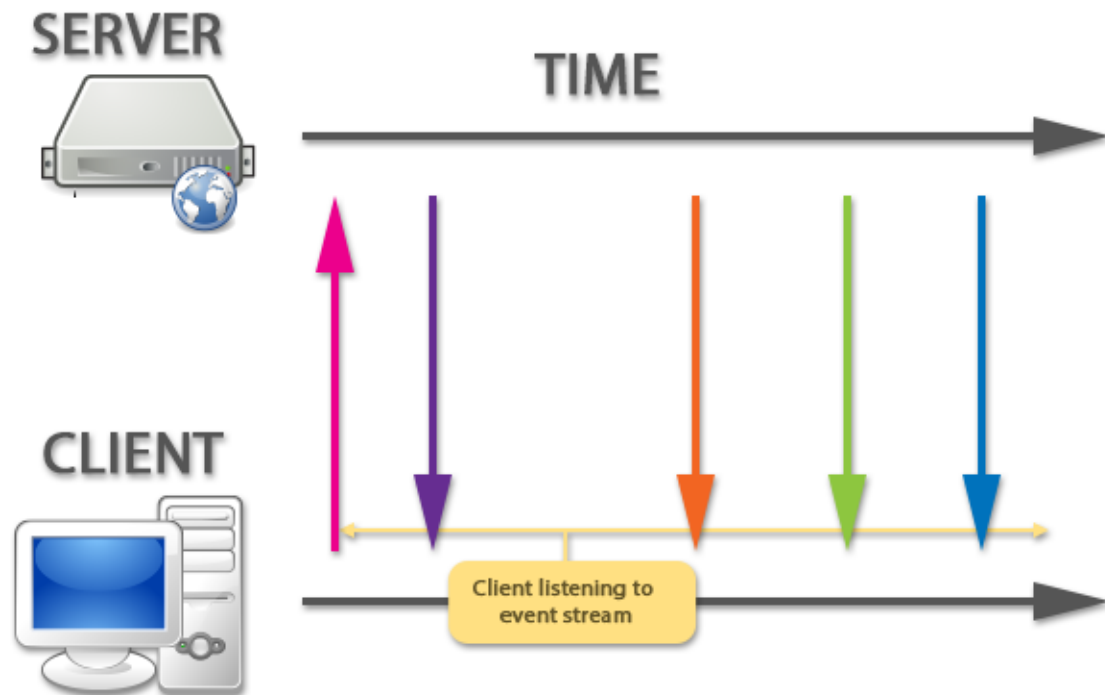
**Figure 3: The WebSocket Protocol.** Each arrow represents a message sent by either server or client. Messages can be sent unrequested in either direction as long as the channel is open

### 2.2.2 Server-Sent Events

Server-Sent Events is a light weight alternative to the WebSocket protocol – it enables a simple model for sending messages from the server to the client without requiring that a request is made. The client establishes a connection to the server, requesting the response in the form of a text/event-stream. The server keeps the connection open and can now send unrequested messages to the client (Vinoski, 2012). See figure 4 for an illustration of the SSE API.

Note that the client cannot send additional messages to the server during the open connection – the client can only listen to an event stream, and not respond through the same stream. Therefore, if the client needs to send additional messages to the server, new HTTP requests have to be sent (using XHR, for example).





**Figure 4: Server-Sent Events.** The first arrow represents the client-server handshake, establishing the SSE channel. The later arrows represent messages sent by server to client using the SSE channel. The client cannot send messages to the server through this channel

## 2.3 Web servers

Since more traditional web servers (Apache, IIS) do not typically have native support for the more recent inventions in the area of real-time communication technology, the experiment applications in this project will run off of Node.js.

### 2.3.1 Node.js

Node.js is a server-side JavaScript environment, supporting long-running server processes (Tilkov & Vinoski, 2010). Node.js contains a built-in HTTP server library, which enables rapid development of custom web servers (node.js).

Node.js is known for being able to provide a platform for scalable, efficient and quick internet applications. This is achieved using a non-blocking event-driven I/O, meaning that time-consuming operations (such as file access, network communication, etc.) do not block the system from handling additional requests (Zhao, Xia, & Le, 2013). Therefore, it is suitable for handling connections from many sources at once.

### 3 Problem formulation

The web browser is rapidly becoming a ubiquitous tool for doing most anything online. In the past, multiple desktop-based applications have been required for performing different tasks, such as real-time voice communication, online gaming or video streaming. With the advent of a number of technologies designed to allow real-time full duplex communication over the web, all of these tasks can now be done directly through a web browser (Qurashi & Anwar, 2012).

However, choosing which of these technologies to use can be a daunting task for a web developer when developing a new web application, as there are many relevant aspects to consider. For example, these technologies differ in terms of capabilities, limitations, browser support, security and performance. Although researching and comparing all of these qualities would be interesting, this thesis will focus solely on performance due to time and scope constraints.

There have been a few studies done on similar subjects in the past. Shuang and Feng (2013) performed a study comparing short polling, long polling, HTTP streaming and the WebSocket protocol. An experiment was conducted to measure one way server push technologies (pushing unrequested data from server to client). They concluded that the WebSocket protocol outperformed the other technologies heavily, in all measures studied.

Pimentel and Nickerson (2012) performed a similar experiment as Shuang and Feng, comparing polling, long polling and WebSocket in a one-way server push environment similar to the experiment described above. Their results also indicate that WebSocket outperforms the other two technologies heavily.

Lubbers and Greco (2010) compare the WebSocket protocol with short polling in an experiment application that updates stock quotes every second. Again, the conclusion is that the WebSocket protocol outperforms polling. In this experiment, a three to one reduction in latency and up to 500 to one reduction in header traffic was observed.

Unlike all of the studies above, this work includes HTML5 Server-Sent Events in the investigation, a technology which appears to be mostly overlooked today. Furthermore, while the related work focus on continuous one-way communication from server to client (also known as server push technologies), this study includes full-duplex communication. That means continuous two-way communication between client and server.

This project aims to answer to central questions:

- Which one of these web technologies perform the best when trying to achieve full duplex real time communication?
- How does network environment affect their performance?

#### 3.1 Delimitations

The technologies chosen for comparison are short polling using Ajax, HTML5 Server-Sent Events and the WebSocket protocol. All of these can be used to achieve or simulate real-time duplex communication through a web browser between the client and web server. The reasoning behind the choice of technologies is that these technologies represent three distinct approaches to achieving real-time communication over the web:

- **Short polling using Ajax** is the oldest approach and gained a lot of popularity due to the Web 2.0 trend in 2005 and 2006. It is also the most unsophisticated way of achieving duplex communication, as it relies solely on the HTTP protocol (Noureddine & Damodaran, 2008)
- **The WebSocket protocol** has received a lot of buzz in the web development world recently, going so far as to being called “A Quantum Leap in Scalability on the Web” in the widely quoted article by Lubbers & Greco (2010). The WebSocket protocol, introduced along with the HTML5 specification, offers full-duplex bidirectional communication over a single socket and is soaring in popularity in many areas of web application development
- **HTML5 Server-Sent Events** (SSE's) actually predate the WebSocket protocol, being implemented in the Opera web browser as early as 2006 and later being standardized as part of the HTML5 Working Draft. Despite this, it has received little attention, perhaps due to the rising popularity of the WebSocket protocol. Criticism has been leveraged against WebSockets, saying they're too heavy duty and introduces a lot of complexity on the client and server ends (Vinoski, 2012). HTML5 SSE's have been said to be a better alternative for light weight and straight-forward server-initiated communication

Besides these, there are many other approaches to achieving real-time communication. Comet and Long Polling represent two older and more traditional methods. There is also a number of web browser plugins available that can efficiently handle real-time communication (such as Adobe Flash, Microsoft Silverlight, etc.). It would be interesting to include these other technologies in the analysis, but due to their fading popularity and time constraints they will not be included in this study. However, they would work well as targets for future research on the subject.

## 4 Method

In order to test and compare the performance of the aforementioned technologies, an experiment was chosen as the methodology to be used because it suits the purpose of the evaluation well. It offers a high level of control while providing the tools necessary to make an accurate evaluation and comparison between the given technologies (Wohlin, Runeson, Höst, Ohlsson, Regnell, & Wesslén, 2012).

The experiment was designed and set up in similar fashion to those used in earlier research on the subject by, amongst others, Agarwal (2012) and Shuang & Feng (2013). Three separate applications (one for each technology) have been created to test the performance, described later in this chapter.

### 4.1 Experiment scope

The problem the experiment aims to resolve is explored in detail in chapter 3 of this thesis, but can be summarized as “How do we determine which technology offers the best performance for real-time communication over the web?” In this context, performance is defined as latency.

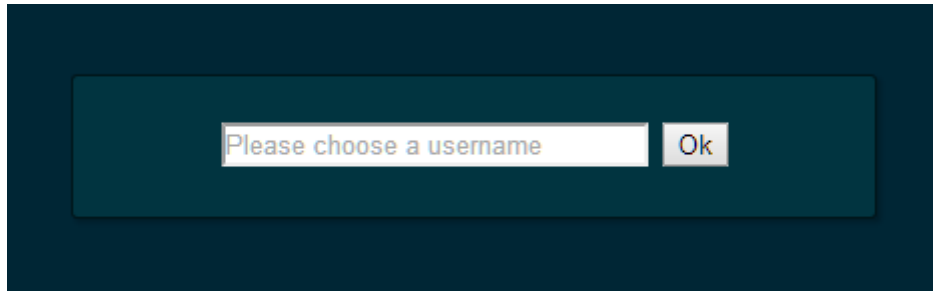
Thus, the goal of the experiment is to analyze different web technologies in order to determine which of them offers the best performance in the context of real-time web communication (Wohlin, Runeson, Höst, Ohlsson, Regnell, & Wesslén, 2012).

### 4.2 Independent variable: the experiment applications

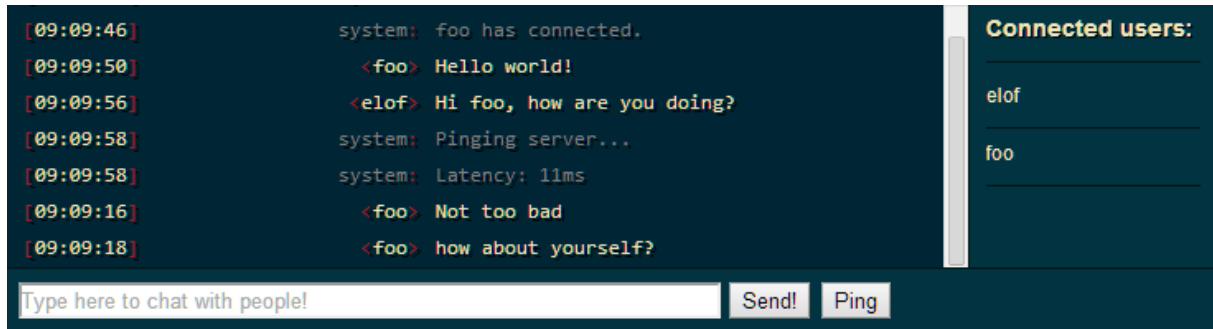
Three separate applications were created to test the performance. “Application” in this context means a combination of server and client software that together function as a tool for instant messaging. The applications are identical in functionality and differ only in the technology used. Thus, there is one WebSocket application, one Ajax application and one Server-Sent Events application. The application, or more specifically the technology behind the functionality of the application, is the independent variable of the experiment (Wohlin, Runeson, Höst, Ohlsson, Regnell, & Wesslén, 2012), i.e. the variable that is manipulated during the experiment.

The functionality of the applications is kept to a very basic level. A typical use case scenario looks like this:

- User opens application through web browser and is presented with a “Choose username” dialog (See figure 5)
- After entering username, user is entered into chat room, where other connected users are listed in a sidebar (See figure 6)
- User can send messages, which are displayed to all other connected clients



**Figure 5: GUI, username input dialog**



**Figure 6: GUI, main client view**

### 4.3 Dependent variables: the performance attributes

To study and compare the effects of using different technologies, a number of performance attributes have been measured. These are the experiment's dependent variables, i.e. what will fluctuate depending on which independent variable being used (Wohlin, Runeson, Höst, Ohlsson, Regnell, & Wesslén, 2012).

The performance attributes that were tested were also chosen based on those used in earlier research on the subject. The interesting attributes are:

- Message time, the time it takes for a message to be sent from a client, handled by the server and displayed for all other connected clients. This can also be called Round trip time, i.e. the time it takes for a message to be sent from the client, handled by the server and displayed back to the client in the public chat room
- The number of HTTP requests required to send a number of messages. This attribute is interesting, since HTTP packets contain a fair amount of header data, thus bandwidth consumption will increase depending on how many HTTP requests are made

### 4.4 Performing the measurements

The performance measurements were executed by sending 100 messages from a client and examining how long it takes for the messages to reach all other connected clients. This was done through timestamps embedded in the messages.

To complement the latency measurement, a performance measurement tool was used to gather data about how many HTTP requests were performed during the message sending

measurement. This tool is the built-in network monitor located in the Google Chrome web browser.

## **4.5 Network considerations**

When discussing the performance of technologies working over the web, the network becomes an important aspect to consider (Shuang & Feng, 2013). To find out which technology operates best under different types of networks, the measurements were constructed in different network environments:

- Server and client running on the same local machine, removing network interference entirely
- Server and client placed on the same local ethernet network, over a 100mbit/s connection
- Server connected to the internet over a fast internet connection and client operating over a wireless 4G connection. These tests were performed during different times to test for any eventual variance in network load

This setup is similar to the one used in the experiment performed by Agarwal (2002).

## **4.6 Alternative methodologies**

In this thesis, experiment was chosen as the methodology to use. However, that is not to say that other methodologies could be equally interesting, depending on which perspective one takes as a researcher.

One problem with choosing experiment as the given methodology is that it only allows us to test a very narrow range of factors that could be important when deciding which technology to use. For example, the experiment will not test the usability of the technologies, usability in this context being “the extent to which a product can be used by specified users to achieve specified goals with effectiveness, efficiency, and satisfaction in a specified context of use” (International Standards Organization, 1998). In this context, the “users” would mainly be application developers, i.e. people utilizing these technologies as tools in their development process. In order to test the technologies from a usability aspect, it’s likely that survey would be a more suitable methodology, targeting developers who have used these technologies actively (Wohlin, Runeson, Höst, Ohlsson, Regnell, & Wesslén, 2012).

Further, due to the nature of any method that makes use of controlled experiments, only a “laboratory view” of the given technologies will be achieved – i.e. an extremely controlled and limited view which is not representative of a real-life context (Wohlin, Runeson, Höst, Ohlsson, Regnell, & Wesslén, 2012). In real-life situations there are a number of factors that may not become apparent in a laboratory setting but are still important to the viability of the technology itself.

Another methodology that could’ve been used is case study, comparing existing implementations of the technologies. This would’ve allowed a comparison for factors other than performance, for example comparing the security features or technical capabilities of the technologies. This would allow for testing in a real-world environment as opposed to a laboratory environment (Wohlin, Runeson, Höst, Ohlsson, Regnell, & Wesslén, 2012).

## **4.7 Other considerations**

### **4.7.1 Hardware specifications**

Another factor which could be interesting in this study, but will not be included due to time constraints, is hardware performance, i.e. how do the specifications of the client, server, and routing machines affect the performance of the applications.

### **4.7.2 Research ethics**

As always when performing experiments including benchmarks there are considerations concerning research ethics. These considerations include the validity of the presented results and the corresponding conclusions. In this experiment, two primary concerns have been identified, following the guidelines presented in the paper “Making Benchmarks Unbeatable” (Cai, Nerurkar, & Wu, 1998):

- 1) Performing the correct and optimized benchmarks – i.e. constructing the experiment in such a way that important and useful results can be extrapolated
- 2) Accurate and honest reporting of results – i.e. reporting truthful and honest results

For the first concern, optimized benchmarks variables were chosen based on earlier research on the subject. Thus, the hope is that these variables will be valid and valuable for this experiment as well.

For the second concern, in their paper Cai et al. (1998) suggest setting up an environment where the benchmark scheme is a dialogue between a vendor (the system designer) and a tester. Because both of these roles will be filled by the experiment conductor, i.e. both creating the testing environment and performing the actual benchmarks will be done by the same person, this is impossible for this experiment.

Instead, the testing applications full source code will be included as appendices to this thesis. That way, the experiment is fully reproducible by anyone who wishes to validate the results. The entire result set will also be included, instead of a cross-section that could theoretically support some specific conclusion. That way, the entire result set will be available for interpretation by outside parties.

## 5 Implementation

In this section, the implementation of this thesis project will be described. First, a literature review will be given, detailing some of the inspirational sources used for this project. After this, a brief description of the pilot study that was performed will be given, followed by a thorough explanation of how the experiment applications were built and how the measurements were performed.

### 5.1 Literature review

This project's design and structure was inspired by a number of different previous projects, scientific papers, websites and applications. Amongst others, this includes the papers and articles mentioned in section 3.1 of this thesis. Aside from the ones listed there, some other sources have been used as inspiration for this project. A few of those sources will be described here.

#### 5.1.1 “WebSockets and Long Polling” by Christian Cromnow

This Bachelor's thesis (Cromnow, 2012) compares long polling to WebSockets through a custom made browser game written in HTML5 and JavaScript.

The experiment uses round trip time as a performance measurement, looking at the time it takes for individual packets to make the trip from client to server and back to the client. This approach to measuring performance and latency was an inspiration in designing the experiment used in this thesis. It also provided the idea of embedding timestamps in the message structure of the application.

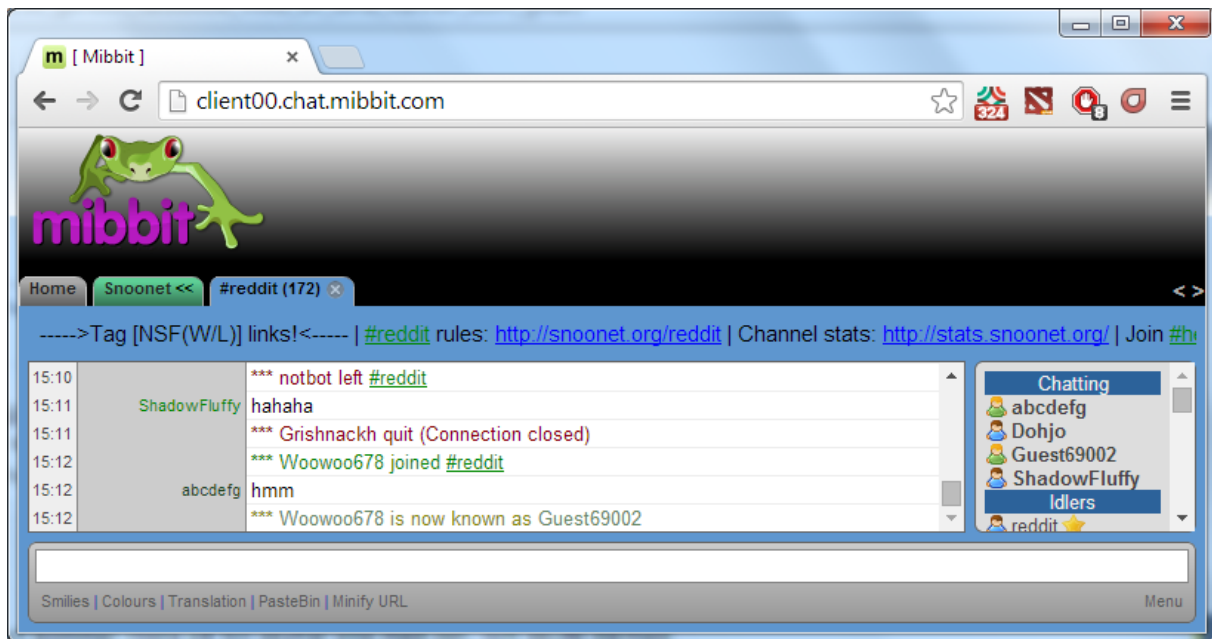
#### 5.1.2 Mibbit – web based text chat application

Mibbit is a text chat application based on the IRC (Internet Relay Chat) protocol (Oikarinen & Reed, 1993). Operation of Mibbit is similar to non-web based IRC clients: the user selects a username, selects an IRC server to connect to and selects which IRC “channel” (chat room) to join. The user can then send and receive messages to other users in the same channel. Mibbit has a few special features (such as automatically creating thumbnails for images when they are linked to in the channel, displaying active chatters and idlers on the sidebar, tabbed channel browsing), but is otherwise very similar in functionality to a regular IRC client.

Technically, mibbit is based on Ajax and repeated polling of an internal server. Mibbit runs a proxy server written in JavaScript that handles and translates IRC connections to an accessible format, which is then polled by the web client. The polling is a form of long polling where connections are kept alive and messages withheld until there are new updates (Moore, 2008). For an explanation of how long polling works, see chapter 2.1.2 of this thesis.

The inspiration garnered from studying mibbit.com was mostly in functionality. There was no investigation of the underlying code of the application, rather the interface and general functionality of the application served as inspiration.





**Figure 7: Screenshot of web-based IRC client Mibbit**

### 5.1.3 Server-Sent Events with Yaws

In this article (Vinoski, 2012), the author presents Server-Sent Events as an alternative to polling to achieve real time communication between server and client. He explains the advantages of SSE's, its standardization process at W3C and finally provides an example using SSE's in YAWS (Yet another web server).

This article is one of few sources dealing with SSE's. SSE's ended up in the back seat of the HTML5 wave as WebSocket received much more attention – this article raised the question of how well this lesser known technology fares against its more popular counterparts.

## 5.2 Pilot study

In order to determine if the chosen methodology could be used to effectively test the different web technologies, a pilot study was conducted. This pilot study functions as a minimal version of the final experiment, with only a small portion of the functionality included in the final experiment.

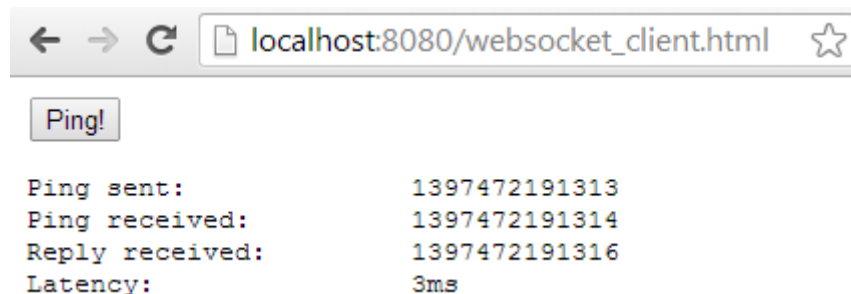
For each technology (WebSocket, HTML5 SSE, Ajax) a small-scale web server was set up. The only function of these web servers is to send a message to a client when a request is received from that client. Included in the message is a timestamp that determines how long the “round-trip” took, i.e. the time for the request to be handled by the server and returned to the requesting client.

The functionality for each application is nearly identical: a button with the label “Ping!” is presented to the user. When the button is pressed, a ping message is sent to the server – when a response is received, an output is displayed to the user, containing 4 values:

- The timestamp for when the message was sent from the client
- The timestamp for when the message was received by the server
- The timestamp for when the response was received by the client
- A calculated latency, achieved by subtracting the last and first timestamp

The messages in all the applications are JSON objects containing properties corresponding to the structure described above.

Figure 8 is a screenshot from the WebSocket pilot client, after the button has been pressed. This output is identical for the three applications.



**Figure 8: Screenshot of WebSocket pilot application after “Ping!” button is pressed**

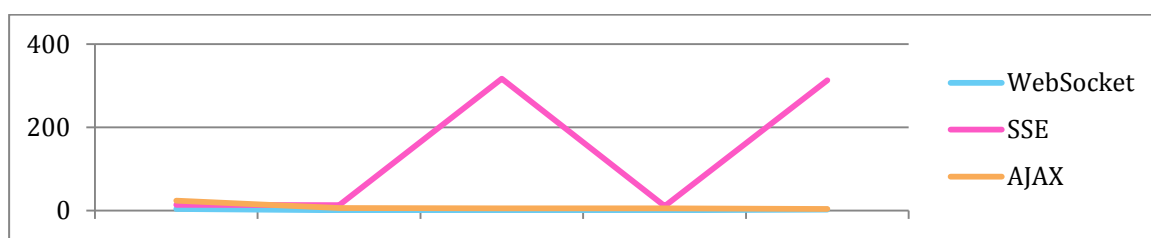
In this way, it is possible to measure the latency for every technology between a single client and the server. From this small-scale experiment we can extrapolate whether a larger scale performance measurement is feasible.

The code for all the server and client applications can be found in Appendix A of this document

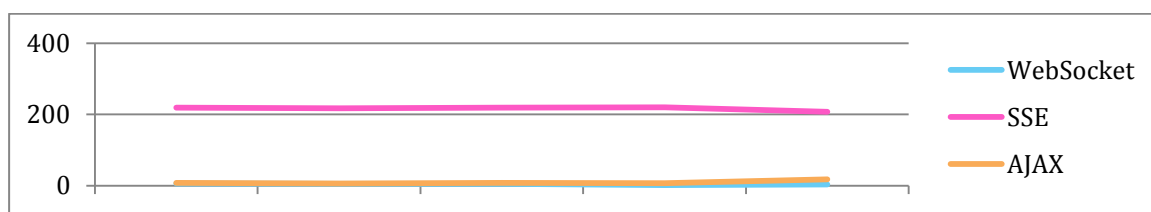
### 5.2.1 Pilot study measurement

In order to test if a larger experiment could garner useful performance information, a set of test measurements were performed. The factor being tested is the latency displayed on screen for each technology. Each technology is also tested under 3 separate network conditions: locally on the same machine, over a wireless LAN and over a 3G connection.

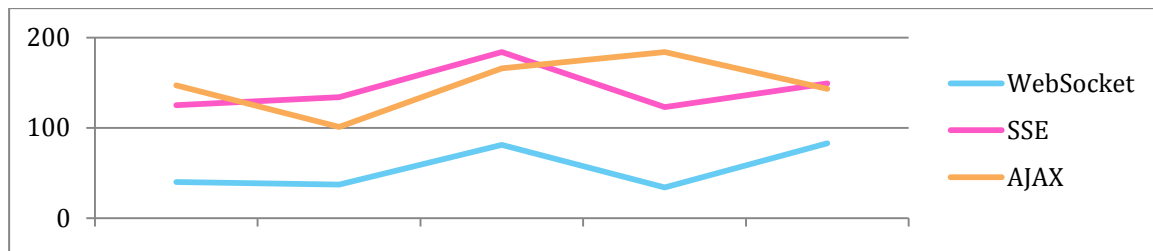
Figure 8, 9 and 10 display the results of the pilot measurement on local machine, LAN and over 3g. The raw data from the measurements can be found in Appendix B of this document.



**Figure 8: Local Machine latency in milliseconds over 5 attempts**



**Figure 9: WIFI LAN latency in milliseconds over 5 attempts**



**Figure 10: 3G latency in milliseconds over 5 attempts**

### 5.2.2 Pilot study discussion

The purpose of the pilot study was not to procure any meaningful results regarding the performance of each individual web technology. Therefore there will be no analysis or conclusions drawn based on the measurements procured. Rather, the intent was to see if it was possible to create a larger experiment which could garnish valuable performance measurements, which could in turn be used to answer the central problem of this thesis.

To this end, the results of the pilot study show that yes, a larger experiment similar in functionality to this smaller study can yield valuable results. The pilot study proves that the involved technologies (WebSocket, SSE's and Ajax) can be used to setup an experiment in which latency is measured. The pilot study also shows that the chosen supporting technologies (the web server Node.js, the scripting language JavaScript and associated framework jQuery, the modules ws and express for node.js, JSON object notation) can be used to create the experiment applications.

Thus, it was demonstrated that the larger experiment can be used to answer the central problem of the thesis.

## 5.3 Building the experiment applications

As mentioned earlier in this thesis, the experiment was conducted by measuring the performance of three distinct web technologies: Ajax, Server-Sent Events (SSE) and WebSocket. To this end, three separate applications were constructed, one for each technology. In this section, an overview of the preparatory work for building the applications will be given, followed by more detailed descriptions about how each individual application was built.

The first step in developing the applications was deciding which tools were to be used. In order to make the measurements as focused and possible and uninfluenced by outside factors, a common toolset was desired, where each application used the same tools. This way, the strengths and drawbacks of each technology could be compared, instead of the qualities of the tools used.

Node.js was chosen as the web server backend. The reason for this was that Node.js offered modules for easy WebSocket development and a real-time paradigm where many tasks can be executed concurrently and the environment could easily be scaled up to handle multiple connections at the same time. Because of this, it was suited especially well for the intended application functionality (real-time messaging between many users) (Zhao, Xia, & Le, 2013).

JavaScript and jQuery were chosen as the front end scripting languages, layered on top of HTML5. Here, the reasoning was more about ubiquitous availability: HTML and JavaScript is well supported natively in all major web browsers, and jQuery is a widely used JavaScript

framework that allows for simplified DOM manipulation and Ajax connections, as well as native JSON manipulation (Dhand, 2012).

Although the techniques used are different for each application, the same basic structure is used for all three applications:

- The back end for each application is a custom web server written in JavaScript and powered by Node.js, using the module express
- The front end for each application is a HTML5 page styled by CSS with scripting in JavaScript and the JQuery framework

The three applications are nearly identical in functionality, as described in section 4.2 of this thesis. The functionality for the applications, a multi-user internet text chat environment, was chosen because it suits the technologies well: measuring real-time communication between clients and server.

The end user will not notice any difference between the applications, except a slightly different delay and responsiveness. The user interface and functions will all remain identical. Again, this is to make sure the measurements stay focused solely on each technology's performance, not other factors.

After these central decisions had been made, work begun on building each individual application. The source code for all three client applications can be seen in appendices C, D and E of this document.

### 5.3.1 Internal message structure of all applications

All the messages sent between client and server, in all three applications, are simple JSON objects. JSON was chosen over other data formats because of its light weight structure, giving it characteristics of small space occupancy and fast transmission speed, which is crucial in a text chat application (Lin, Chen, Chen, & Yu, 2012). It can also be transmitted as plain text easily, and is widely supported for manipulation in both native JavaScript and jQuery, on both Node.js and web browser platforms.

The following is an example of a message sent from server to client in the WebSocket application, in the form of a plain text JSON object:

```
{
  author: "elof",
  type: "chatMessage",
  content: "hello world",
  clientSent: 1397570293045,
  serverSent: 1397570293046,
  clientReceived: 1397570293050,
  roundTrip: 5
}
```

This message is also an example of another common practice that is used in all applications: passing a message along through multiple steps, adding attributes along the way. In this example, the message originated from a single client, containing only the `author`, `type`, `content` and `clientSent` attributes. When it was received on the server, the server appended the `serverSent` attribute, containing the timestamp detailing when the server received the message. Finally, the message was sent from the server to all other connected

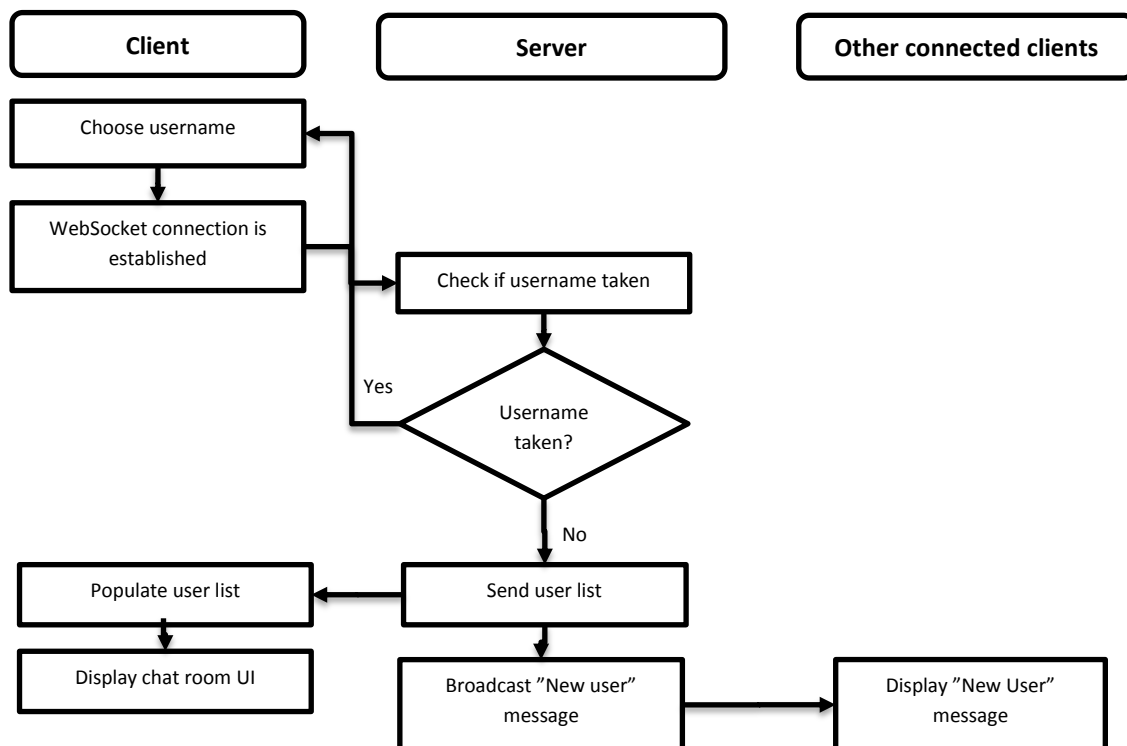
clients, who in turn appended the final attributes `clientReceived` and `roundTrip`, to determine how long the message took to send. Using this practice, it is possible to determine how long the message took to be processed on each step of the way.

### 5.3.2 The WebSocket application

What sets the WebSocket protocol apart from the other technologies described, is that it allows for a continuous connection where both server and client can send unrequested messages to each other for as long as the connection is kept open. The source code for the WebSocket application can be found in appendix B of this document.

#### 5.3.2.1 Connection procedure

Here, the process of a new user connecting to the server is described. See figure 11 for an overview of the connection procedure, followed by an explanation including code examples.



**Figure 11: How a new connection is handled using the WebSocket protocol**

On the server side, an internal user list is maintained with every active WebSocket connection. This list contains the details of the WebSocket connections, along with a custom property containing the username of each connection. This means that each username is unique.

When a client first opens the application, they are asked to select a username. Here, the first challenge with using WebSocket was encountered: how do we check if the username is already in use without first creating a connection? The obvious solution would be to make a separate request using Ajax to fetch the user list before establishing the WebSocket connection.

However, since the purpose of this application was to test the WebSocket protocol only and not Ajax, a different approach was designed: the WebSocket connection is immediately established and the client sends a message containing the username to the server, but the user is not added to the internal user list on the server side. Instead, the server checks if the username is taken – if it is not, the user is added to the user list with the desired username. If it is taken, the user is informed through the WebSocket connection, which is then immediately closed. On the client side, the user is prompted to select a new username and a new WebSocket connection is established towards the server.

The following code excerpt shows the code for checking if the username is taken on the server side, and what happens if it is:

```
if(messageObject["type"] == "firstConnection") {
    var nameTaken = false;

    for(var i = 0; i < connectedClients.length; i++) {
        if(connectedClients[i].username == messageObject["username"]) {
            nameTaken = true;
            ws.send(JSON.stringify({
                "type": "usernameTaken",
                "username": messageObject["username"]
            }));
            break;
        }
    }
}
```

`connectedClients` is a list of socket objects containing each active connection. Among many other attributes, they have a custom `username` attribute containing the username.

The following code excerpt shows what happens on the server side when the username isn't taken. The variable `ws` contains the socket information for the connected client. The `broadcast` method is a custom method to send messages to all currently connected clients.

```
if(!nameTaken) {
    ws.username = messageObject["username"];

    var userlist = [];
    connectedClients.forEach(function(e) {
        userlist.push(e.username);
    });

    ws.send(JSON.stringify({
        "type": "connectionSuccessful",
        "userlist": userlist
    }));

    connectedClients.push(ws);

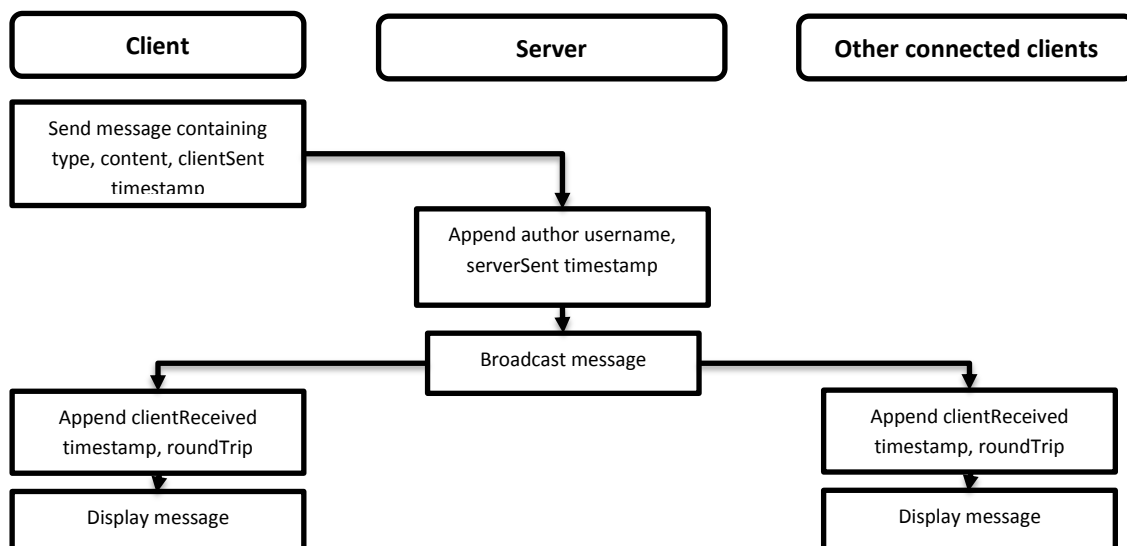
    broadcast({
        "type": "userConnected",
        "username": ws.username
    });
}
```

### 5.3.2.2 Handling messages from the client

There are three types of messages that can be sent from the client to the server: `chatMessage`, `ping` and `firstConnection`.

A `chatMessage` contains a chat message from a single user to be distributed to all other connected clients. Along with the original message, the author of the message (stored in the `username` property of the `WebSocket` object in the client list) and the timestamp for when the server received the message are appended to the object. See figure 12 for an illustration of the path a chat message takes through the application.

```
if(messageObject["type"] == "chatMessage") {  
  var reply = messageObject;  
  reply["serverSent"] = Date.now();  
  reply["author"] = ws.username;  
  broadcast(reply);  
}
```



**Figure 12: How a chat message is handled**

A `ping` message is a simple object containing only the `type` property. When the server encounters it, it immediately replies with a message containing only the `pingReply` type. There is no timestamp. Instead, the latency is calculated on the client side by measuring the time it takes to get a reply. The `JSON.stringify()` method is used to convert a simple JavaScript object to a JSON string..

```
if(messageObject["type"] == "ping") {  
  ws.send(JSON.stringify({  
    "type": "pingReply"  
  }));  
}
```

The third type, `firstConnection`, is sent when the user first connects. It has two properties, `type` and `username` where `username` contains the desired username for the

connecting client. What happens when this message is received is described section 6.1.2.2 of this thesis.

### 5.3.2.3 Handling messages from the server

All DOM manipulation on the client side (i.e. the adding and removing of elements and text in the HTML hierarchy) is done through jQuery.

There are six different messages that the server can send to the client: `usernameTaken`, `connectionSuccessful`, `chatMessage`, `userConnected`, `userDisconnected`, and `pingReply`.

`UsernameTaken` is sent when the user attempts to connect using a username that's occupied. When this message is received, the client displays a "Username already taken" message to the user, and the WebSocket connection is closed.

```
$("#intro form").append('<p>Username in use! Try a different one.</p>');  
window.ws.close();
```

`ConnectionSuccessful` is sent when the user has chosen a username which is not already taken. Aside from the type, this message also contains the `userlist` property, containing the users already connected. This is then used to populate the user list in the user interface.

```
function populateUserlist(list) {  
  list.forEach(function (e) {  
    $("#connectedUsers").append('<li>' + e + '</li>');  
  });  
}
```

`UserConnected` and `userDisconnected` happen when another user joins or leaves the chat room. The messages contain a type property and the username of the person joining or leaving. When this message is received by the client, the username is added or removed from the user list and a notice is displayed in the chat window. `SystemMessage` is a custom method used for displaying status messages in the chat window.

```
$("#connectedUsers").append('<li>' + username + '</li>');  
systemMessage(username + " has connected.");
```

`ChatMessage` is sent by the server when another user sends a message. It contains a number of different properties, including 3 separate timestamps (one when the user sent the message, one when the server passed the message on to all other connected clients and one when the client received the message). It also contains the username of who wrote the message, as well as the message content. When this message is received, it is displayed in the chat window.

### 5.3.3 The Server-Sent Events (SSE) application

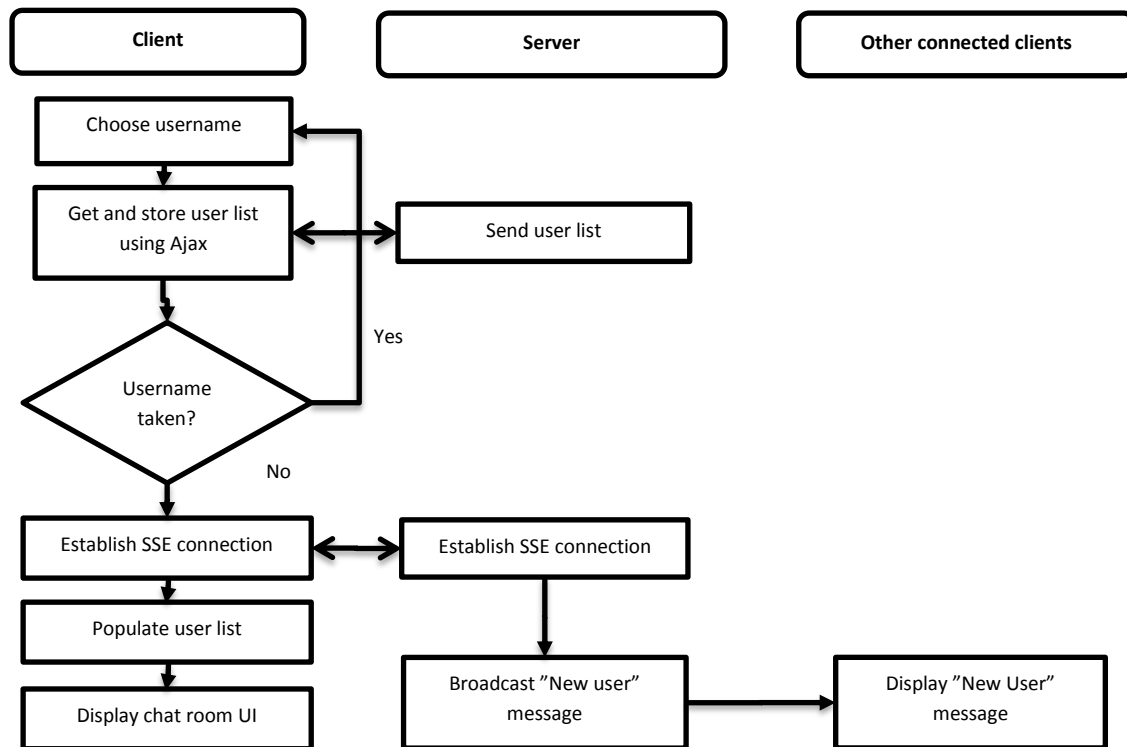
SSE's can, similarly to the WebSocket protocol, be used to establish a continuous connection between client and server. However, unlike the WebSocket protocol, the client cannot send unrequested messages to the server through the SSE channel. The connection, once established, is a one-way communication channel where the server can push messages to the client, but not the opposite (see figure 4 for an illustration of this structure). However, full duplex communication can be achieved by layering other communication techniques on top



of the SSE connection. The source code for the SSE application can be found in appendix C of this document.

### 5.3.3.1 Connection procedure

Because SSE is a one way communication protocol once a connection is established, there is no way for the client to fetch the user list through the SSE connection. However, other communication techniques can be used to fetch the user list before the SSE connection is even established.



**Figure 13: How a new connection is handled using the SSE protocol**

In the SSE application, this is done through Ajax. Thus, the connection procedure begins by the user choosing a username, the client fetching the user list using jQuery's Ajax methods and checking if the chosen username is available. If it is, set up the SSE connection. If it's not, display a message to the user. The following code excerpt shows what happens after the user list has been fetched from the server.

```
$.get('/userlist', function(data) {
    window.userlist = data;
    if(data.indexOf(window.username) == -1) {
        setupSSE(window.username);
    } else {
        usernameTaken();
    }
});
```

The SSE connection is set up on the client side by creating an `EventSource` object pointing to the server. `EventSource` is a JavaScript object that exists natively in all browsers

supporting the SSE standard. The application uses URL parameters to also send the username of the client connecting. This way, the username can be added to the internal connected client list on the server side.

On the server side, the application sends a particular set of HTTP headers that indicate to the browser that it is a SSE connection. These include `content-type: text/event-stream` and `connection: keep-alive`. This means that the connection will never be closed, unless the client disconnects manually.

Finally, the server broadcasts a message saying a user has connected to all the connected clients.

```
app.get('/es/:username', function (req, res) {
  console.log("NEW USER!!!");
  req.socket.setTimeout(Infinity);
  res.writeHead(200, {
    'content-type': 'text/event-stream',
    'connection': 'keep-alive',
    'access-control-allow-origin': '*'
  });

  res.username = req.param("username");

  connectedClients.push(res);

  broadcast({
    "type": "userConnected",
    "username": res.username
  });
});
```

### 5.3.3.2 Handling messages on the server side

Message handling is done similarly to the WebSocket application. The messages take the same format; JSON objects with timestamps appended on each step of the journey from client to server to all other clients (see section 6.1.1).

However, there is one key difference. As mentioned earlier, SSE's do not support duplex communication, so to achieve duplex communication in the chat application another communication technique must be used to send messages from client to server. In this application, Ajax is used for this purpose.

Thus, the server needs to be able to handle both the SSE connection (to send messages to connected clients) and regular HTTP requests (to receive messages from clients). The server listens to POST requests sent to the URL `"/message"`. The two message types it can receive from connected clients are `"ping"` and `"message"`. In express, POST parameters are passed through the `request.body` property. `broadcast` is a custom method that sends a message to all currently connected clients. `sendTo` is a custom method that sends a message to a user with a certain username. The following code excerpt shows what happens on the server side when it receives a POST request to the url `"/message"`.

```
app.post('/message', function(req, res){
  if(req.body.type == "chatMessage") {
    var reply = req.body;
```

```

        reply["serverSent"] = Date.now();
        broadcast(reply);
    } else if(req.body.messageType == "ping") {
        sendTo(req.body.messageAuthor, "pingReply", "");
    }
    });

```

Here we encounter another challenge caused by SSE's lack of two-way communication functionality. When the client sends a “ping” message, it expects a “pingReply” message back, in order to determine latency. However, since messages are received on the server side through POST but broadcast to all connected clients using SSE, there is no easy way to send a reply to only the same specific client. Thus, we must send the author username of the message along with the POST request, and then manually compare the author to the list of connected clients' usernames and finally send the reply through the proper client socket. The following code excerpt shows the custom `sendTo` method, which is used to send SSE messages to a client with a specified username.

```

function sendTo(username, ev, content) {
    for(var i = 0; i < connectedClients.length; i++) {
        if(connectedClients[i].username == username) {
            send(ev, content, connectedClients[i]);
        }
    }
}

```

`Send` is another custom helper method, used to send SSE messages. SSE messages need to be formatted according to the Event stream format (w3.org, 2012), a format that must take the following appearance:

```

event: chatMessage
data: {"username": "elof", "content": "hello world"}

```

Thus, the method `send` looks like this.

```

function send(ev, content, responseObject) {
    responseObject.write("event: " + ev + "\n");
    responseObject.write("data: " + content + "\n\n");
}

```

### 5.3.3.3 Handling messages on the client side

Messages sent from the server to the client are sent through the established SSE connection. Thus, we set up listeners on the `EventSource` object created during the setup phase, listeners that perform specific actions based on the type of event that was received. The two types of events the application listens to are `pingReply` and `message`.

`Message` is either a chat message from a connected client or a user connected message from the server. If it is a `userConnected` message, the message is simply displayed in the UI along with the username of the connected user. When it is a `chatMessage` message, an additional timestamp is appended, the final roundtrip time is calculated (by subtracting the first timestamp from the last timestamp) and the message is displayed in the UI.

The following code excerpt shows the message event listener. `chatMessage()` and `userConnected()` are simple GUI methods to display messages in the UI.

```
source.addEventListener("message", function(e) {
    var messageObject = JSON.parse(e.data);

    if(messageObject["type"] == "chatMessage") {
        chatMessage(messageObject["author"], messageObject["content"]);
        messageObject["clientReceived"] = Date.now();
        messageObject["roundTrip"] = (messageObject["clientReceived"] -
messageObject["clientSent"]);
        console.log(messageObject);
    }

    if(messageObject["type"] == "userConnected") {
        userConnected(messageObject["username"]);
    }
}, false);
```

The second event type, `pingReply`, is received after the user has sent a ping request. When the client sees this event, it subtracts the stored ping value (a timestamp created when the ping request was sent) from the current timestamp and displays the latency in the UI to the user. `SystemMessage` is a simple GUI method to display system messages in the chat UI.

```
source.addEventListener("pingReply", function(e) {
    var diff = (Date.now() - window.latestPing);
    systemMessage("Latency: " + diff + "ms");
}, false);
```

### 5.3.4 The Ajax application

The Ajax application represents the older method of building real-time web applications. It uses the traditional HTTP request/response structure and there is no channel whatsoever to send unrequested messages from server to client. The Ajax solution works by the client repeatedly polling the server, checking for updates. The source code for the Ajax application can be found in appendix D of this document.

#### 5.3.4.1 Connection procedure

The connection procedure poses no significant challenge in the Ajax solution – it works similarly to the SSE and WebSocket applications. After the user enters a username, the client sends a POST request to the server containing said username. If the username is not in the server user list, an “ok” message is sent back to the client which in turn initializes the primary loop and displays the chat room UI to the user. The following code excerpt shows what happens when the client connects to the chat room.

```
function setupAjax(username) {
    window.username = username;
    $.post('/connect', {"username": username}, function(data) {
        if(data == "ok") {
            $("#intro").hide();
            $.get('/userlist', function(data) {
                window.userlist = data;
                populateUserList(data);
            });
        }
    });
}
```

```

window.onbeforeunload = function() {
    $.post('/userdisconnect', {"username": username});
};

setInterval(primaryLoop, 200);
} else {
    usernameTaken();
}
});
}

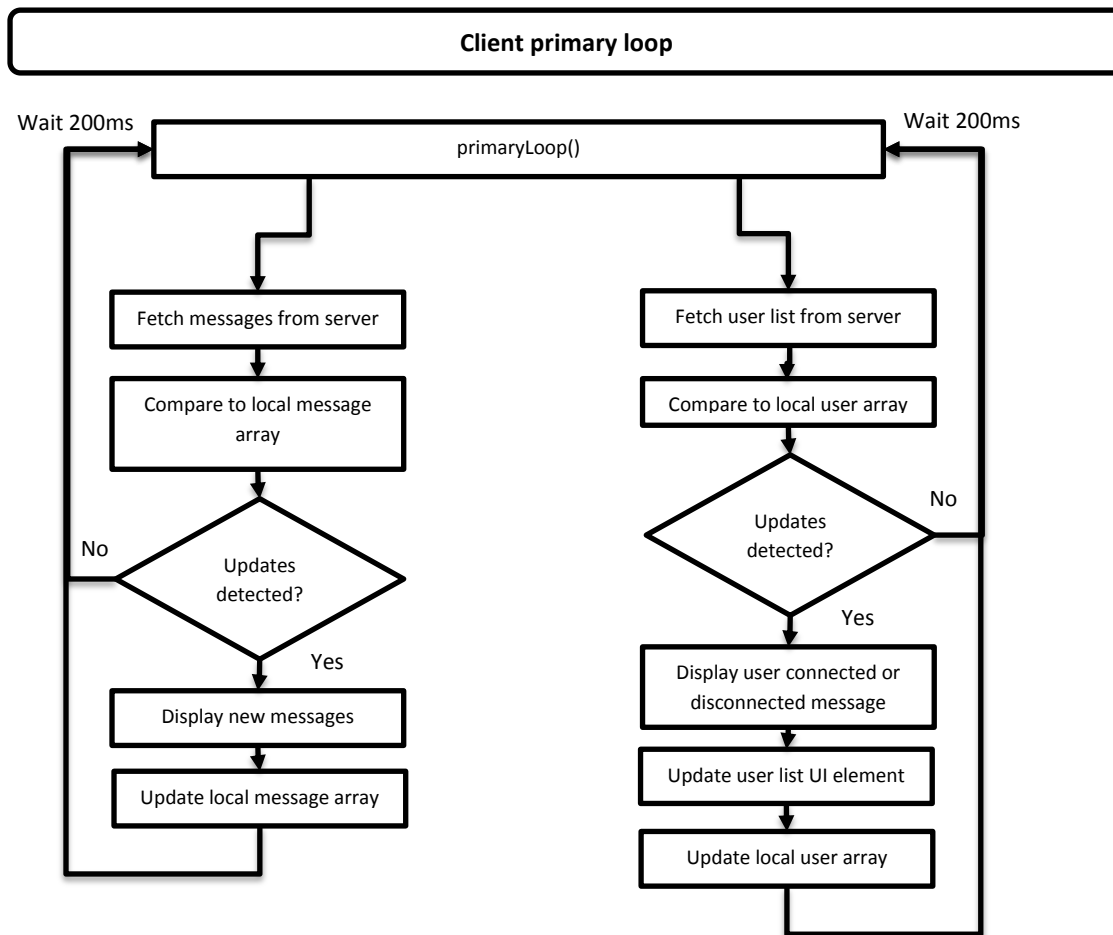
```

When the user disconnects (i.e. closes the browser tab or window) another POST request is sent to the server, containing the username of the user disconnecting. The server removes the username from the internal client list.

#### 5.3.4.2 Client side primary loop

Since the Ajax solution is strictly bound to a request/response structure, there is no way to automatically detect when a message is posted by another user. Therefore, the client must repeatedly poll the server and compare to stored data, checking for any eventual updates. It does this in a looped function, executing every 200 milliseconds.

Connection and disconnection of other clients also prove to be challenging events to deal with when using an Ajax response/request structure. There is no way for the server to tell each individual client when a user connects or disconnects, as there is no way to send unrequested messages through traditional HTTP. Therefore, the client must repeatedly poll the server about connected users to detect when a disconnection or connection occurs. See figure 14 for an illustration of what happens in every cycle of the primary client loop.



**Figure 14: Primary client side loop**

The following code excerpt shows what happens in the primary loop on the client application. `userConnected` and `userDisconnected` are custom methods, displaying a message to the user and interacting with the user list UI element.

```

function primaryLoop() {
  $.get('/fetchmessages/'+window.username, function(data) {
    if(data.length > 0) {
      sortMessages(data);
    }
  });

  $.get('/userlist', function(data) {
    for(var i = 0; i < data.length; i++) {
      if(window.userlist.indexOf(data[i]) == -1) {
        userConnected(data[i]);
      }
    }

    for(var i = 0; i < window.userlist.length; i++) {
      if(data.indexOf(window.userlist[i]) == -1) {
        userDisconnected(window.userlist[i]);
      }
    }
  });
}

```

```
}
```

The following code excerpt shows what happens when the messages are fetched from the server. `Window.messages` is the global variable containing the client message array. The second if-statement in the `sortMessages` method compares `clientSent` timestamp of the very last message in the local array to the very last message in the fetched messages array. If they are different, then both arrays are compared using the custom `getDifference()` method, the new messages are displayed using the custom `parseMessages()` method and are added to the local message array.

```
function sortMessages(fetchedMessages) {
  if(window.messages.length == 0) {
    window.messages = fetchedMessages;
    parseMessages(fetchedMessages);
    return;
  }

  if(window.messages[(window.messages.length-1)]["clientSent"] !=
  fetchedMessages[(fetchedMessages.length-1)]["clientSent"]) {

    var difference = getDifference(window.messages, fetchedMessages);
    window.messages = window.messages.concat(difference);
    parseMessages(difference);
    return;
  }
}
```

#### 5.3.4.3 Handling messages from the client

On the server side, the handling of messages from the client is straight forward. The server listen to POST requests sent to the “/message” URL. When one is received, the message is appended with a timestamp called `serverReceived` and the message is placed in an internal message array.

This array is limited to 10 messages, containing only the latest messages to conserve memory and the amount of data that needs to be sent back and forth.

The following code excerpt shows what happens when the server receives a POST request on the “/message” URL. `TrimArray` is a custom method that cuts an array down to the specified length, taking from the top.

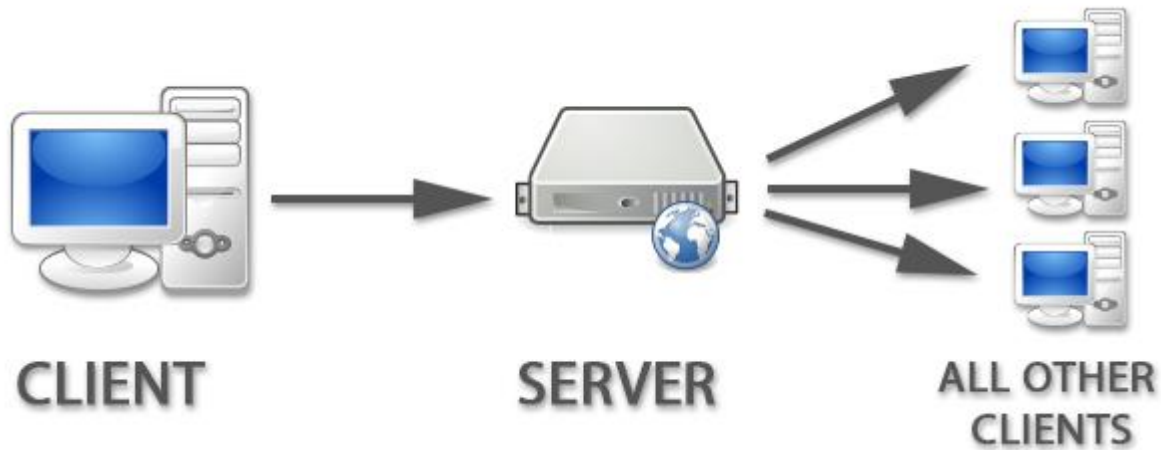
```
app.post('/message', function(req, res) {
  var message = req.body;
  message["serverReceived"] = Date.now();
  trimArray(messages, messageLimit);
  messages.push(message);
});
```

## 5.4 Performing the measurements

Two things were measured on all three applications: message latency and number of HTTP connections. In this chapter the procedure of performing these measurements will be described.

### 5.4.1 Measuring message latency

The first performance attribute being measured is message latency. This is defined as round trip time, the time it takes for a message to complete the path from the client sending a message, to the server, to all other connected clients. See figure 15 for an illustration of a completed round trip.



**Figure 15: The complete path a message takes before being measured**

As mentioned earlier, message latency is stored within each message: timestamps are appended to the message at each node in the path. So all that needs to be done to measure latency is to record messages and scan their timestamps.

This was done using an additional script layered on top of the original applications. The script works by automatically entering form data into the chat input and automatically submitting that data. It performs this 100 times with a 500 millisecond delay between each message.

Because of the message structure described in earlier sections of this thesis, the latency information is already included in the message objects being sent. Therefore, the only thing that needs to happen for message latency to be logged is to store the messages in an internal array. The script then presents these stored messages to the user.

The messages being sent contain a randomized string, varying in length and size. There's no great variance here, as the purpose of the experiment was not to test the performance of very large or very small messages, although this would be interesting as a future project.

The following code excerpt shows the function used to send messages *n* times. `randomString()` is a function returning a random string between 5 and 25 words long, with word lengths being between 3 and 6 characters each.

```
function runTest(n) {  
  var i = 0;  
  
  var interval = setInterval(function() {  
    $("#controls form #message").val(randomString());  
    $("#controls form").submit();  
    i++;  
  }, 500);  
}
```



```

    if(i >= n) {
        clearInterval(interval);
    }
}, 200);
}

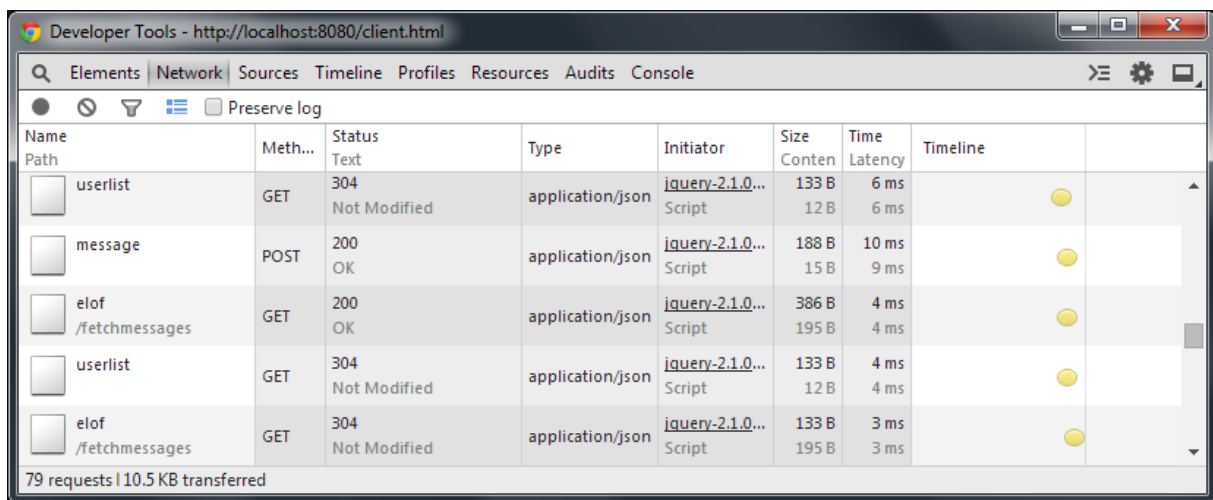
```

The measurement script can be found in its entirety in appendix F of this document.

#### 5.4.2 Measuring number of requests made

The second performance attribute being measured is the number of requests required for the previously described operation, namely sending 100 messages from client to all other clients. This is an interesting attribute because each http request and response carry with them unnecessary header information (Shuang & Feng, 2013) which increase bandwidth consumption and thus directly affect latency. Note that only the sending of the messages is measured, not the connection procedure. Thus, measurement starts once both sending and receiving clients have finished connecting and entered the chat room.

This measurement was made using the built-in development tools in the Google Chrome web browser. These tools make it possible to record all HTTP requests and responses on an active web page. See figure 16 for an illustration of what the Chrome web developer tools look like in action.



The screenshot shows the Chrome Developer Tools Network tab for the URL `http://localhost:8080/client.html`. The 'Network' tab is selected, and the 'Preserve log' checkbox is checked. The table below represents the data shown in the screenshot.

Name Path	Meth...	Status Text	Type	Initiator	Size Conten	Time Latency	Timeline
userlist	GET	304 Not Modified	application/json	jquery-2.1.0... Script	133 B 12 B	6 ms 6 ms	
message	POST	200 OK	application/json	jquery-2.1.0... Script	188 B 15 B	10 ms 9 ms	
elof /fetchmessages	GET	200 OK	application/json	jquery-2.1.0... Script	386 B 195 B	4 ms 4 ms	
userlist	GET	304 Not Modified	application/json	jquery-2.1.0... Script	133 B 12 B	4 ms 4 ms	
elof /fetchmessages	GET	304 Not Modified	application/json	jquery-2.1.0... Script	133 B 195 B	3 ms 3 ms	

79 requests | 10.5 KB transferred

**Figure 16: Chrome developer tools used for measuring HTTP request**

These measurements were performed both on the sending side and the receiving side, leading to six data sets: two for each technology.

## 6 Results and analysis

All measurements were conducted on machines with the following specifications:

- The server machine is a stationary PC with an Intel Core i5-2500K CPU running at 3.30GHz. The machine has 12GB ram and runs Windows 7 Ultimate. The application server software runs on node.js version 0.10.25. This machine also acts as a client on the local machine measurements
- The client machine on the LAN and 4G measurements is a Lenovo E530 Thinkpad laptop, with an Intel Core i7-3612QM CPU running at 2.10GHz. The machine has 8GB RAM and runs Windows 7 Home Premium, 64-bit edition. The client tests were performed in the browser Google Chrome version 34.0.187.137

As explained earlier in this thesis, two kinds of measurements were made:

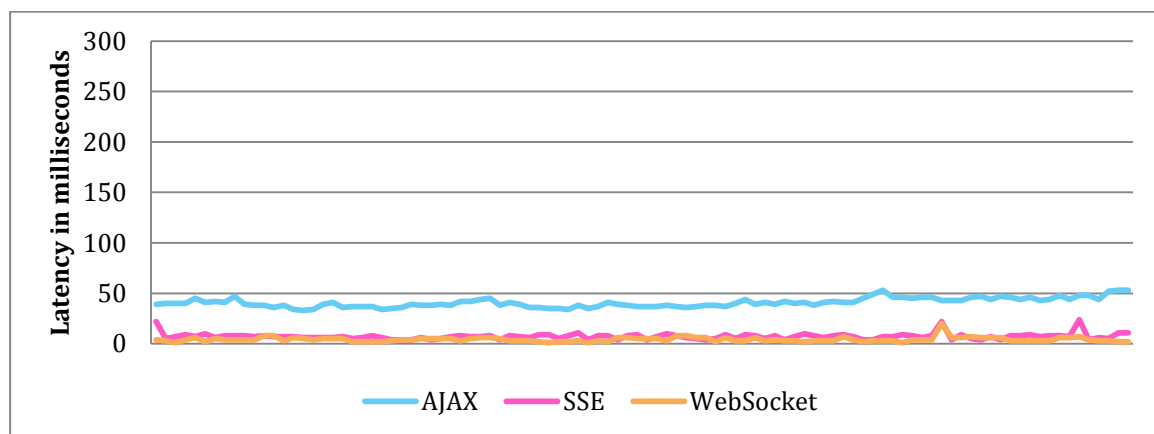
- Message latency, measured over 100 messages and three network conditions
- Number of HTTP connections required to send 100 messages. Since this remains static over any network condition, only one set of measurements were made

In this chapter, the results of the measurements will be presented and compared. First, the results of the latency measurements will be presented separated into the different network environments. An analysis comparing the results between the different technologies follows. Finally, the HTTP request measurements will be presented and discussed.

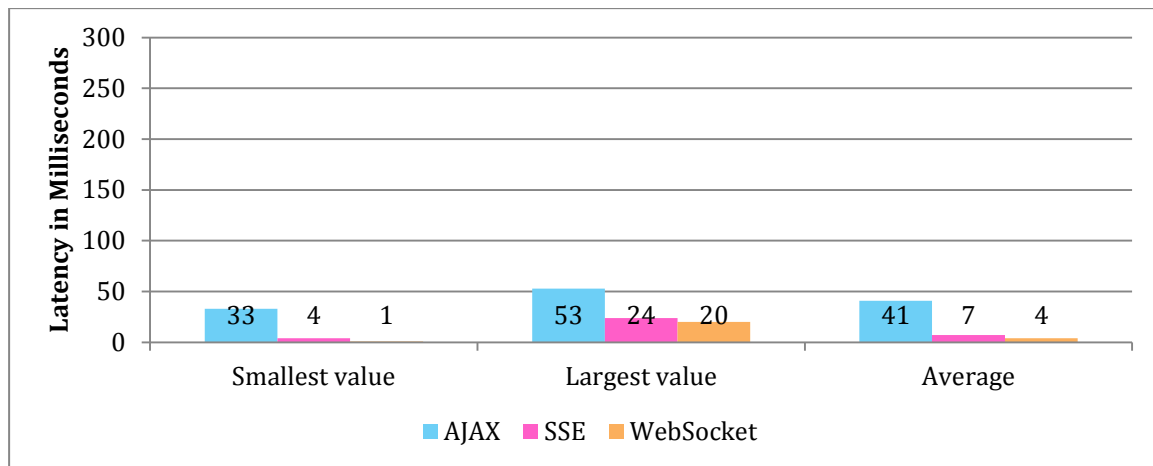
### 6.1 Local machine latency measurements

The local machine measurements were performed with both the server software and the client software running on the same local machine. In practice, this meant running the Node.js server software in a Windows command prompt, while accessing the client through a web browser.

Figure 17 and 18 demonstrate latency over 100 messages being sent and notable values from this measurement.



**Figure 17: Latency while sending 100 messages over local machine**



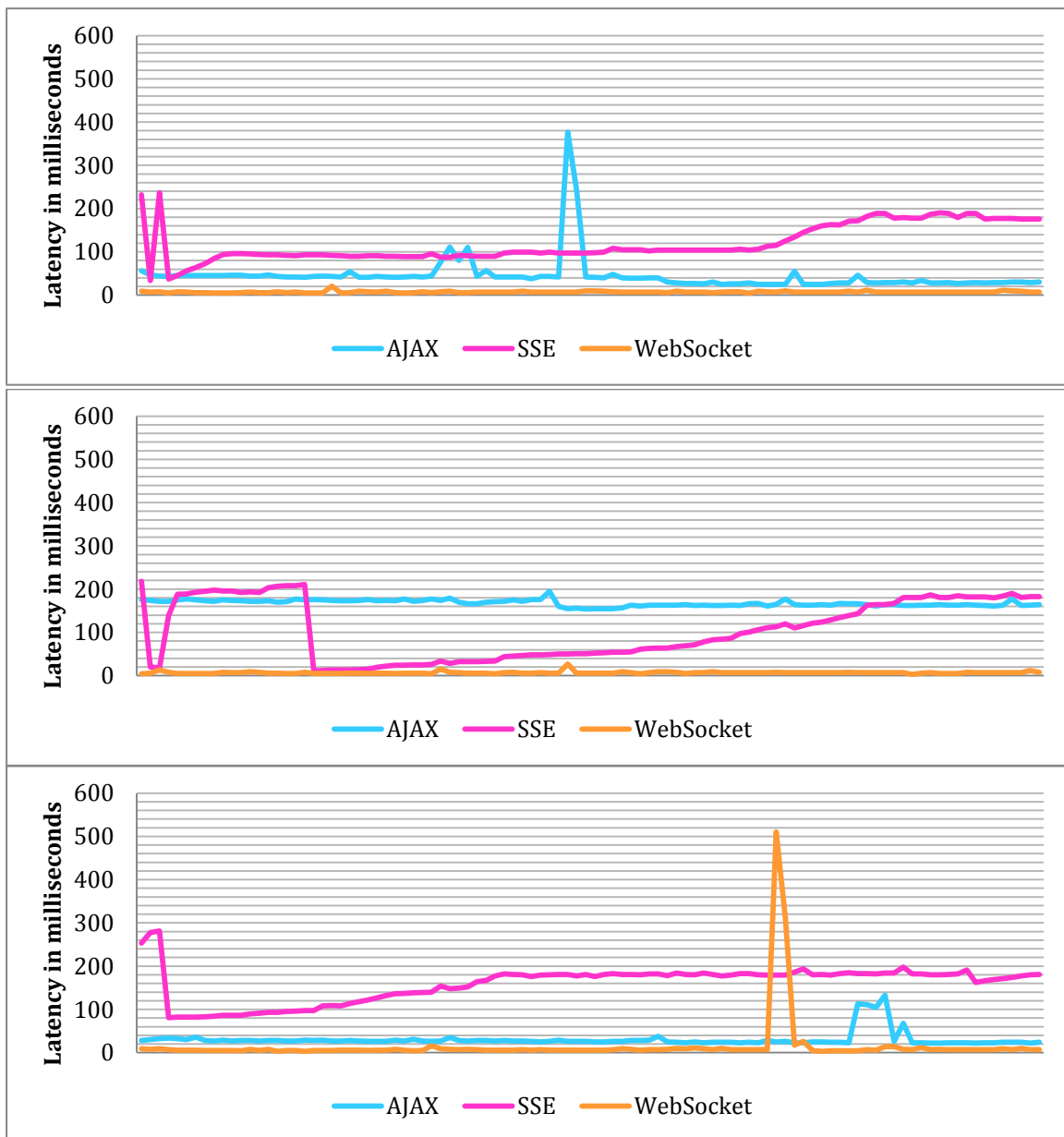
**Figure 18: Notable values for the local machine measurements**

The local machine measurements can be found in their entirety in Appendix G of this document.

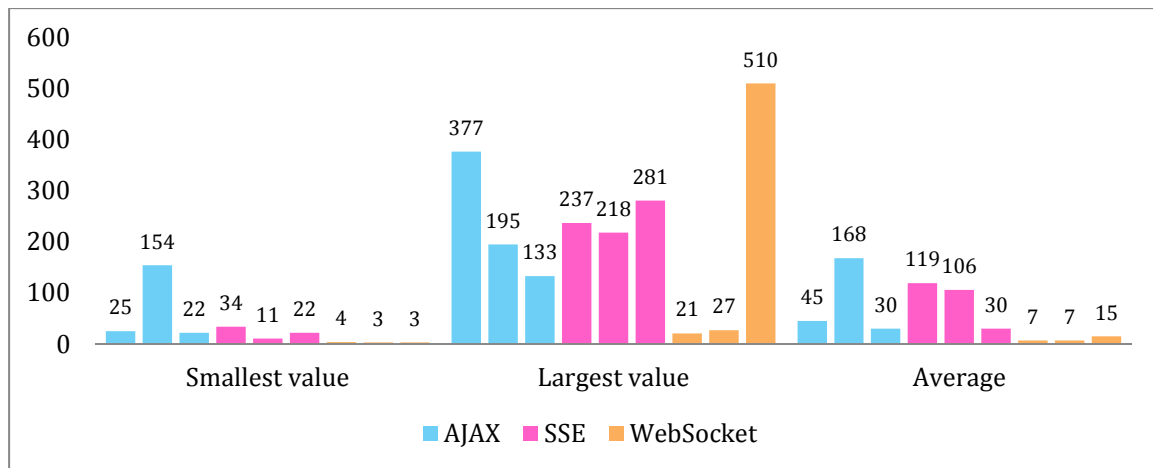
## 6.2 LAN latency measurements

The LAN measurements were made over a high speed local area network. The server computer was under a wired 100mbit/s connection to the network, while the client computer was under a wireless 100mbit/s connection, placed in close proximity to the wireless router (within 10 meters).

Three separate LAN measurements were made, leading to a result set of 300 message round trip times in total. Figure 19 and 20 demonstrate latency over 100 messages being sent and notable values from these measurements.



**Figure 19: 3 charts showing latency over time while sending 100 messages over 100mbit/s Wifi LAN, from three separate measurements**



**Figure 20: Notable values from the three LAN measurements**

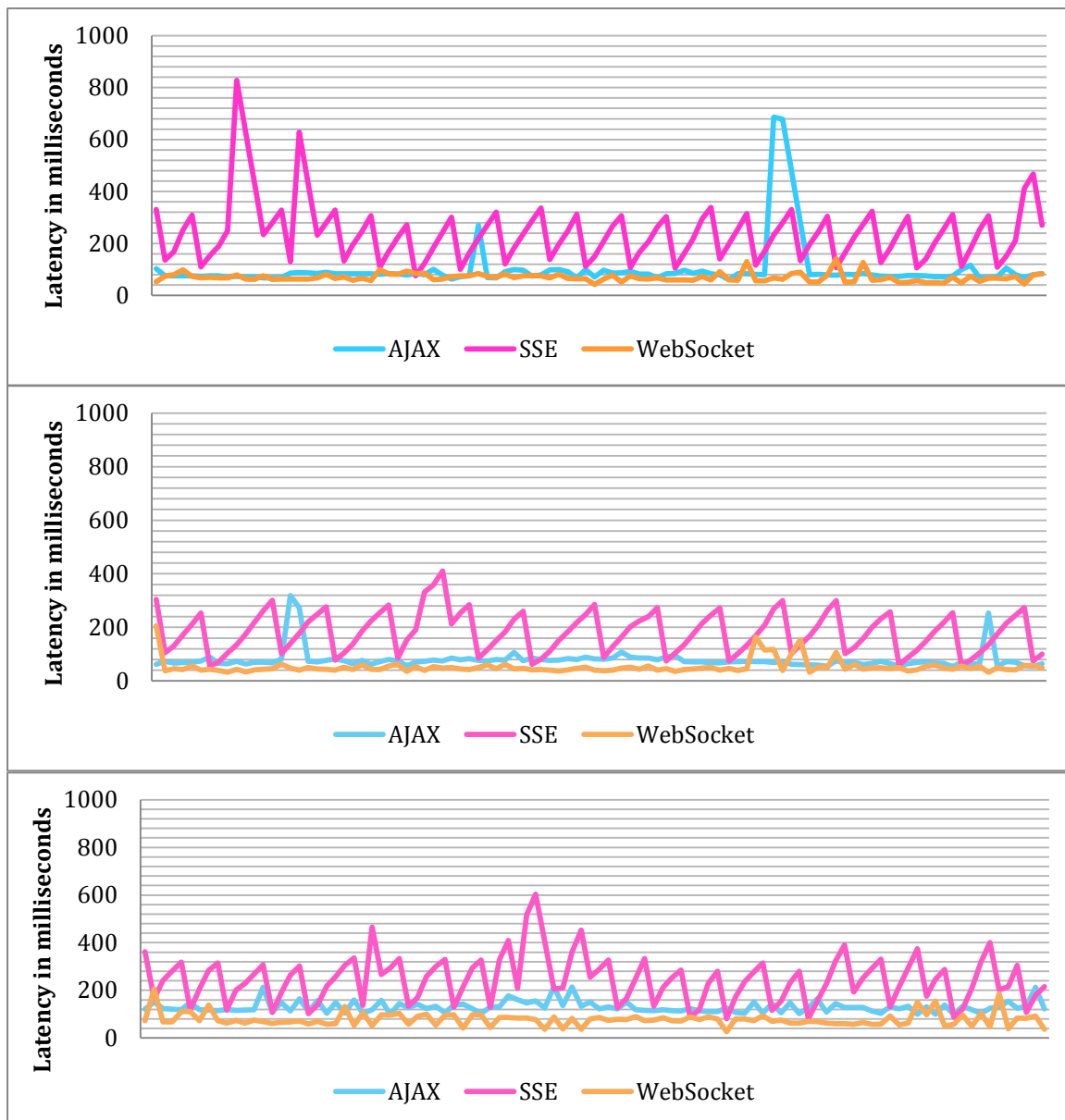
The LAN measurements can be found in their entirety in Appendix H of this document.

### 6.3 4G latency measurements

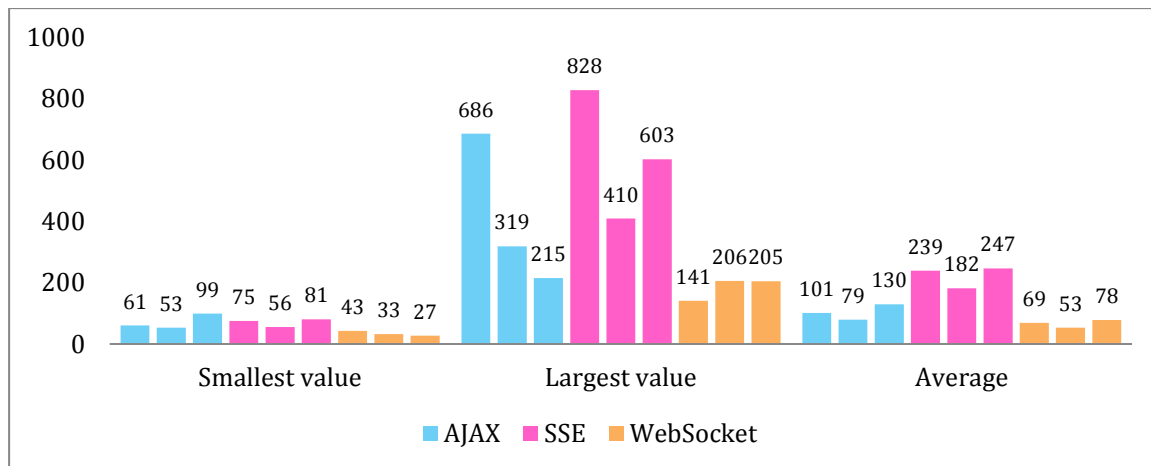
The 4G measurements were made in three separate rounds, to get some variance in 4G network activity.

The first round was made on a Thursday at 14:00. The second round was made on a Friday at 11:00. The third round was made on a Thursday at 22:00. Although there's no data available on the network stress during these times, the three separate measurements will hopefully provide some variance in network activity during measurement.

Figure 21 and 22 demonstrate latency over 100 messages being sent and notable values from these measurements.



**Figure 21: Three charts showing latency over time while sending 100 messages over 4G network**



**Figure 22: Notable values for the three 4G measurements**

The 4G measurements can be found in their entirety in Appendix I of this document, along with exact dates and times of the three measurements.

## 6.4 Latency analysis

In this chapter the result data presented in the previous sections will be analyzed and summarized, separated into each technology being tested.

### 6.4.1 Short polling via Ajax

The results of the Ajax Short Polling measurements were surprising in their consistency. On local machine, all latency values were between 31 ms and 50 ms, which is on the high side but the values were very consistent. Likewise, LAN and 4G measurements were consistent with average values of 45 and 101 with a few spikes, but largely even during the course of the measurements.

On local machine, Ajax fares a bit worse than the other technologies due to its short polling structure. Since the short polling interval was set to 200ms, i.e. fetching new messages from the server five times every second, the latency is noticeably larger. The fact that latency is shorter than the polling interval is because sending messages is completely asynchronous, i.e. not bound to polling interval. Thus, messages can be sent faster than they're retrieved, and the sending interval likely matched up with the retrieving interval, making the latency shorter than 200 ms.

Ajax outperforms SSE on 4g and performs about as well as 4G on LAN. This was a surprising result, considering the relative age of the two technologies and the way they function. Short polling works by repeatedly fetching resources from the server and updating the local message queue – this would appear to be a very inefficient way to receive updates, but performs quite well and consistently under all three network conditions.

### 6.4.2 Server-Sent Events

On Local Machine, the SSE application performed about as well as the WebSocket application with an average latency of 7, between 4 and 24 ms.

With an average latency of 119, 106 and 30 on the three LAN measurements, the SSE application is outperformed by both the Ajax and the WebSocket applications. The same is

true for the 4G measurements. With average latencies of 239, 182 and 247 ms the SSE application performs well under the 4G and WebSocket applications.

On top of this, the SSE applications results are the most inconsistent. With large spikes of upwards of 300 ms on LAN, the latency seems to increase progressively during the course of the measurement. This would suggest that the SSE channel cannot handle the pace of the messages being sent, but chokes during the experiment resulting in worse latency.

On 4G, the results are even more inconsistent. With spikes upwards of 800 MS, the latency rises and falls in large spikes and dips during the course of the experiment. Again, this seems to suggest the channel cannot deal with the amount of traffic being sent across the network.

Of all the technologies tested, the server-sent events seem to be the most sensitive to network conditions.

### **6.4.3 WebSocket**

The WebSocket application performs the best overall out of the three applications.

With an average of 4 ms on local machine, with values ranging between 1 and 20, the latency is both consistent and very low – about on par with SSE.

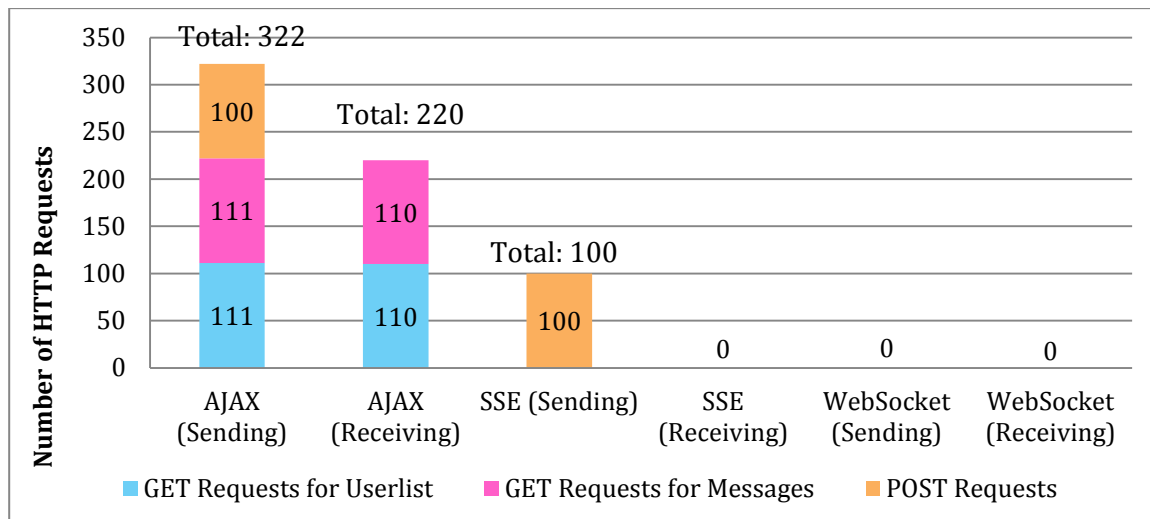
It is on the LAN measurements the WebSocket application really stands out. With average latencies of 7, 7 and 15 across the three measurements and very consistent results, between 4 and 27 ms on every value, the results are both very low and very consistent. The WebSocket application outperforms the other two applications by a large margin.

On 4G, the WebSocket application performs very well. With average latencies of 65, 53 and 67 ms on the three measurements, the WebSocket application performs about twice as well as the Ajax application and about 4 times better than the SSE application. On top of this, the WebSocket application provided very consistent results. There were a few spikes where the latency was doubled for short periods, but aside from that the latency was very even during the course of the measurement.

## **6.5 Number of HTTP connections**

This measurement targeted the number of HTTP connections required to send 100 messages. Due to the functionality of the technologies tested, these results vary greatly. Figure 23 demonstrates the amount of HTTP requests required to send 100 messages, data scraped while performing the latency measurements. The data is divided into 3 categories, depending on the type of HTTP request being made, and the measurements are performed both on the sending side and the receiving side.





**Figure 23: Number of HTTP requests required to send 100 messages**

How many HTTP requests are needed by the various applications vary greatly – this is due to how the technologies function.

The WebSocket protocol functions by opening a bidirectional non-HTTP channel for communication. Thus, there are no HTTP requests made on either the receiving side or the sending side after the initial handshake has been completed. Since only sending the messages was measured, not the chat room connection procedure, the resulting number of HTTP connections on both the client side and the receiving side are zero.

Similarly to the WebSocket protocol, the Server-Sent Events function by opening a continuous channel between client and server. However, since this channel is unidirectional (the server can send unrequested messages to the client, but not vice versa), messages being sent need to be sent through a different route: as HTTP POST request using Ajax. Thus, the receiving client is at zero HTTP connections while the sending client is at exactly 100 HTTP POST requests.

The Ajax application stands out. Not only do all messages being sent have to be posted through HTTP POST requests, the process of fetching new messages has to be done through short polling. Thus, we end up with a large number of HTTP requests, most of which contain no new information, on the receiving side. The polling interval in this experiment is set at 200ms – 5 requests are made each second. On top of fetching new messages, an updated user list is also fetched at the same rate, leading to 4 requests being made each second, simply to keep the chat application updated.

These results would suggest that the Ajax application is much more bandwidth inefficient than the other applications, due to the large number of HTTP requests necessary to maintain normal operation. Even during times when nothing at all is happening in the chat room, the client must repeatedly poll the server several times per second, checking for updates. This leads to a large increase in bandwidth due to HTTP request headers and responses.

## 7 Conclusion

In this chapter, the analysis from the previous chapter will be summarized and put in relation to the problem formulation of this thesis. The results will be further discussed, with recommendations and advisement to any web developer considering these technologies. Finally, a look at the possibilities for future work will be given.

### 7.1 Summary

This project set out to make the choice of web technology easier for web application developers wanting to achieve real-time duplex communication. To do this, an experiment was conducted to answer two central questions:

- Which one of these web technologies perform the best when trying to achieve full duplex real time communication?
- How does network environment affect their performance?

To answer these questions, the experiment was performed on 3 applications, identical in functionality but differing in technology used to achieve real-time communication.

The experiment showed that WebSocket outperformed both Short Polling using Ajax and Server-Sent Events. Over a 4G connection, WebSocket performed about 4 times better than SSE's and 2 times better than short polling over Ajax. The same factor of performance was observed over LAN connections. On local machine, WebSocket performed about on par with SSE and around twice as good as Ajax, but this situation is the least interesting in relation to real world situations.

As well as showing poor performance over LAN and 4G, SSE's proved to perform very inconsistently, with large spikes and dips in latency. This noisy performance suggests the SSE channel cannot adequately keep up with the traffic generated by the experiment over networks, which would mean SSE's are very sensitive to network conditions.

The experiment further showed that short polling using Ajax kept surprisingly good performance, despite the technology's age and structure. With mediocre but acceptable latencies, the Ajax application proved to be very consistent with even and stable latencies across measurements.

What constitutes good performance on the web depends on the usage area. Different applications require different performance to provide an acceptable use experience. For application types such as the experiment application, text chat, latency is not a hugely noticeable factor by an average user. However, considering the prevalence of mobile devices and applications requiring touch surfaces, it might be interesting to look at performance in relation to touch devices. In their study, Ng et al. (2013) write:

---

*In summation, it would appear that achieving a touch-to-display latency of 20ms would be sufficient to ensure the majority of users do not perceive it, and thus would see no performance degradation, when input is restricted to tapping. When direct manipulation is employed, latencies down to 2.38ms are required to alleviate user perception when dragging*

---

Thus, both Ajax and SSE would fall outside of the acceptable spectrum when using touch devices.

Put in relation with the project's goal, it is evident that the WebSocket protocol provides both the most consistent and the best performance across all three network conditions compared to the other two technologies. On top of this, the WebSocket protocol also carries with it no extra bandwidth usage in the form of HTTP requests, in contrast to the bandwidth hungry short polling technology.

## 7.2 Discussion

As explained in chapter 3 of this thesis, this project's goal was to make the choice of which web technology to use easier for a web developer. This choice depends heavily on what needs and requirements the application being developed has. For example, it is becoming increasingly common for applications to target mobile devices – these devices often rely on 4G connections to the internet, which tend to be more unstable and less powerful than other kinds of networks. In this respect, WebSocket would seem like the optimal choice, performing the best over 4G by far, compared to the other technologies.

Mobile devices often rely on touch interfaces, i.e. interaction by tapping, dragging or scribbling with your fingers on the screen. This mode of interaction is much more sensitive to latency than traditional interaction (Ng, Annett, Dietz, Gupta, & Bischof, 2014) (Jota, Ng, Dietz, Dietz, & Wigdor, 2013), which would also suggest WebSocket is an optimal choice.

Additionally, 4G connections tend to be limited in their bandwidth availability. Thus, low bandwidth usage is usually desirable for mobile applications. In this regard, WebSocket also seems like a better choice over the other two technologies; a continuous WebSocket connection requires no additional HTTP requests to keep the connection alive, leading to no extra traffic in the form of HTTP headers and responses.

However, one area where SSE's might be the superior choice over WebSocket is when dealing with a situation where full duplex communication is not required. In a scenario where a server needs to send continuous updates to the client, but the client never needs to send anything to the server (also known as server push), SSE's might be the right technology for the job considering it was this they were explicitly designed for (Vinoski, 2012). This would also depend on factors like server and client CPU usage, which unfortunately were not part of this study.

Regarding short polling via Ajax, this technology might be desirable depending on the browser support requirements of the application being developed. As this study did not take browser support into consideration, no hard recommendations or conclusions can be drawn in this area, but considering the age and widespread usage of this technology in comparison to its more modern counterparts SSE and WebSocket, it would seem likely that short polling via Ajax has far more widespread browser support.

With this in mind, it is the hope of the researcher that this study will prove useful in a larger context, should any aspiring developer worry about performance of real-time communication technologies when getting into the field of web development – a field which is quite saturated with new technologies, usage areas, platforms and environments.

Regarding the ethical aspects of this project, a few concerns were outlined in section 4.7 of this thesis. These concerns include making sure the correct and optimized benchmarks, as

well as making sure results were truthfully and honestly reported. To mitigate these concerns, care was taken in picking which attributes were measured and how they were measured. This was based on earlier research done on the subject, in the hope that following in the footsteps of earlier researchers would lead to accurate and correct benchmarks.

Further, all source code and measurement data are included in appendices to this document. This way, the experiment can be recreated fully, so if anyone wanted to ensure the validity of the measurement data, the experiment is fully repeatable.

Because the experiment was performed by a single person and no instances of personal information were used, personal research ethics were not a factor in this project.

### **7.3 Future work**

There are many possibilities for future work in this field, both in the short and in the long term.

In the short term, simply repeating the experiment while looking at different factors would prove quite valuable. One such factor would be browser support, as mentioned earlier in this thesis. Browser support is of significant relevance when developing new web applications, as it completely changes the tools available to you as a web developer. There is a large range of browsers in use today, varying heavily depending on region of the world, target demographic, whether the application is going to be used in the workplace or at home, etc. Thus, it would be very interesting to see which browser supports which technology, and how well they support it. This would also include looking at different platforms – for example, Google Chrome on mobile might support a technology in the Windows client but not in the Android client. Browser and platform support would play a big role when choosing which technology to use for a new application.

Another factor that would be interesting to investigate more thoroughly would be bandwidth consumption. This experiment scraped the surface of this factor by looking at number of HTTP requests required for each technology. However, this metric alone does not give us the full picture: there might be many packets sent “behind the scenes” that do not take the form of HTTP requests. A study investigating these using, for example, a packet sniffer such as Wireshark or similar software would be very relevant. This way, it would be possible to find out exactly how much bandwidth was needed for each technology, which again impacts the decision of what technology to choose for a new application quite heavily.

Finally, it might be interesting to look at things like CPU stress generated by each technology. It’s likely that more heavy-duty solutions like the WebSocket protocol require significantly more processing on the client and server sides than SSE’s and short polling. Looking at this in a new experiment may prove very valuable.

On a larger scale, it would be interesting to repeat a similar project but using a different methodology. For example, a case study performed in a live situation would yield valuable results. This way, we would get measurements from a real world situation instead of a laboratory view. Measurements could be conducted on a number of different enterprise solutions being used to service real users. These measurements could then be used to draw conclusions more closely linked to real world applications of the different technologies.

Another opportunity for further development would be to take the experiment applications devised for this experiment and continue building on them, into a full-fledged chat

application. This experiment proves that it's at least possible to use the technologies tested to create usable chat real-time chat applications. However, continuing development on these applications would require taking many other factors into account, such as making sure they work under very heavy load with many thousands connected users, adding new features such as multiple chat rooms, private messaging, etc. and perhaps adding on new functionality such as voice or video communication.

With the advent of powerful real-time communication capabilities as well as other features and functions of browsers and web standards, the possibilities for new application development are nearly endless. Combining real-time communication with things such as HTML5 3D and 2D canvas drawing, the HTML5 video and audio APIs, LocalStorage and other new features of HTML5, it is now possible to create almost any type of application directly on the web. We've already seen powerful multiplayer games, such as BrowserQuest, implementing WebSocket as its communication technology. Given time, I think we will move more and more towards an entirely browser-based computing experience, and real-time communication capabilities are an important feature of facilitating this development.

And as new features are being standardized by the HTML Working Group, web browser developers need to keep experimenting with new features and new capabilities. Browser vendors are historically the ones who have pushed web technology forward, with standardization following. This is a trend that needs to continue on all fronts of application development.

## References

- Agarwal, S. (2012). Real-Time Web Application Roadblock: Performance Penalty of HTML Sockets. *2012 IEEE International Conference on Communications (ICC)*, (pp. 1225-1229). Ottawa, Canada.
- Alvestrand, H. (2012). *Overview: Real Time Protocols for Browser-based Applications*. Retrieved 03 17, 2014, from <http://tools.ietf.org/html/draft-ietf-rtcweb-overview-03>
- Anderson, T. (2012). Long Polling with AJAX: A mobile network offloading technique. Bachelor's thesis. Skövde, Sweden: University of Skövde.
- Cai, J.-y., Nerurkar, A., & Wu, M.-Y. (1998). Making benchmarks uncheatable. *Computer Performance and Dependability Symposium, 1998. IPDS '98* (pp. 216-226). Durham, NC: IEEE.
- Cromnow, C. (2012). Websockets and Long Polling: For network communication in situations of high traffic and real-time requirements. Bachelor's thesis. Skövde, Sweden: University of Skövde.
- Dhand, R. (2012). Reducing Web Page Post Backs through JQuery AJAX Call in a Trust Based Framework. *2012 International Conference on Computing Sciences* (pp. 217-219). Phagwara, India: IEEE.
- Fette, I., & Melnikov, A. (2011). *The WebSocket Protocol*. Retrieved January 27, 2014, from Internet Engineering Task Force (IETF) RFC 6455: <http://tools.ietf.org/html/rfc6455>
- Garret, J. J. (2005, February 18). *AJAX: A New Approach to Web Applications*. Retrieved January 27, 2014, from Adaptive Path LLC: <http://www.adaptivepath.com/ideas/ajax-new-approach-web-applications/>
- International Standards Organization. (1998). Ergonomic requirements for office work with visual display terminals. *ISO 9241-11:1998*. International Standards Organization.
- Jota, R., Ng, A., Dietz, P., Dietz, P., & Wigdor, D. (2013). How Fast is Fast Enough? A Study of the Effects of Latency in Direct-Touch Pointing Tasks. *CHI '13 Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (pp. 2291-2300). New York: ACM.
- Lin, B., Chen, Y., Chen, X., & Yu, Y. (2012). Comparison between JSON and XML in Applications Based on AJAX. *2012 International Conference on Computer Science & Service System (CSSS)* (pp. 1174-1177). Nanjing: IEEE.
- Loreto, S., Saint-Andre, P., Salsano, S., & Wilkins, G. (2011). *Known Issues and Best Practices for the Use of Long Polling and Streaming in Bidirectional HTTP*. Retrieved February 3, 2014, from Internet Engineering Task Force (IETF) RFC 6202: <http://tools.ietf.org/html/rfc6202>
- Lubbers, P., & Greco, F. (2010). *HTML5 Web Sockets: A Quantum Leap in Scalability for the Web*. Retrieved February 5, 2014, from WebSocket.org: <http://www.websocket.org/quantum.html>

- Moore, J. (2008, January 4). *Mibbit: Ajax-based IRC Client*. Retrieved April 14, 2014, from Ajaxian.com: <http://ajaxian.com/archives/mibbit-ajax-based-irc-client>
- Ng, A., Annett, M., Dietz, P., Gupta, A., & Bischof, W. F. (2014). In the blink of an eye: investigating latency perception during stylus interaction. *CHI '14 Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (ss. 1103-1112). New York: ACM.
- node.js. (n.d.). *About Node.js*. Retrieved May 12, 2014, from nodejs.org: <http://nodejs.org/about/>
- Noureddine, A., & Damodaran, M. (2008). Security in web 2.0 application development. *Proceedings of the 10th International Conference on Information Integration and Web-based Applications & Services* (pp. 681-685). Linz, Austria: ACM.
- Oikarinen, J., & Reed, D. (1993). *Internet Relay Chat Protocol*. Retrieved April 14, 2014, from Internet Engineering Task Force (IETF) RFC 1459: <http://tools.ietf.org/html/rfc1459.html>
- Pilgrim, C. (2013). An investigation of usability issues in AJAX based web sites. *AUIC '13 Proceedings of the Fourteenth Australasian User Interface Conference* (ss. 101-109). Darlinghurst: ACM.
- Pimentel, V., & Nickerson, B. G. (2012). Communicating and Displaying Real-Time Data with WebSocket. *IEEE Internet Computing*, 16(4), 45-53.
- Qurashi, U. S., & Anwar, Z. (2012). AJAX based attacks: Exploiting Web 2.0. *2012 International Conference on Emerging Technologies (ICET)* (pp. 1-6). Islamabad: IEEE.
- Shuang, K., & Feng, K. (2013). Research on Server Push Methods in Web Browser Based Instant Messaging Applications. *Journal of Software*, 8(5), 2644-2651.
- Tilkov, S., & Vinoski, S. (2010, November 01). Node.js: Using JavaScript to Build High-Performance Network Programs. *IEEE Internet Computing*, 14(6), pp. 80-83.
- Vinoski, S. (2012). Server-Sent Events with Yaws. *IEEE Internet Computing*, 16(5), 98-102.
- Wohlin, C., Runeson, P., Höst, M., Ohlsson, M. C., Regnell, B., & Wesslén, A. (2012). *Experimentation in Software Engineering*. Berlin: Springer-Verlag Berlin Heidelberg.
- World Wide Web Consortium (W3C). (2012). *Server-Sent Events*. Retrieved May 12, 2014, from W3C Candidate Recommendation: <http://www.w3.org/TR/eventsource>
- Zhao, S. M., Xia, X. L., & Le, J. J. (2013). A Real-Time Web Application Solution based on Node.js and WebSocket. *Advanced Materials Research*, 816-817, 1111-1115.

Server icon by RRZEicons, Wikimedia Commons (2012), accessed 2014-02-18, available at <http://commons.wikimedia.org/wiki/File:Server-web.svg>, Creative Commons Attribution-Share Alike 3.0 Unported license

Client icon by Ysangkok, Wikimedia Commons (2006), accessed 2014-02-18, available at [http://commons.wikimedia.org/wiki/File:Computer\\_n\\_screen.svg](http://commons.wikimedia.org/wiki/File:Computer_n_screen.svg), GNU Lesser General Public License



## Appendix A – Pilot application source code

### websocket\_server.js

```
var WebSocketServer = require('ws').Server,
    wss = new WebSocketServer({port: 1234});
var express = require('express');

var app = express();

app.use(express.static(__dirname));

var port = process.env.PORT || 8080;

app.listen(port, function() {
    console.log("Listening on " + port);
});

wss.on('connection', function(ws) {

    ws.on('message', function(message) {
        var msg = JSON.parse(message);

        if(msg['type'] == 'ping') {
            var reply = msg;
            reply["type"] = "pingReply";
            reply["pingReceived"] = Date.now();
            ws.send(JSON.stringify(reply));
        }

    });
});
```

### websocket\_client.html

```
<!doctype html>
<html>
  <head>
    <script type="text/javascript" src="../jquery-2.1.0.min.js"></script>
    <script type="text/javascript">
$(document).ready(function() {
    window.ws = new WebSocket("ws://" + document.domain + ":1234");

    ws.onopen = function(event) {
        $("#output").html("Connected!");
        $("button").css("visibility", "visible");
    };

    ws.onmessage = function(event) {
        var msg = JSON.parse(event.data);
        console.log(msg);
        if(msg['type'] == 'pingReply') {
            msg['replyReceived'] = Date.now();
            $("#output").html("Ping sent: \t\t" + msg["pingSent"] + "\nPing
received: \t\t" + msg["pingReceived"] + "\nReply received: \t" +
msg["replyReceived"] + "\nLatency: \t\t" + (msg["replyReceived"]-
msg["pingSent"])) + "ms");
        }
    }

    $("button").click(function() {
        ws.send(JSON.stringify({
```

```

        "type": "ping",
        "pingSent": Date.now()
    }));
});

});
</script>
<body>
<div>
    <button style="visibility: hidden">Ping!</button>
    <p id="output" style="white-space: pre; font-family:
monospace">Loading...</p>
</div>
</body>
</html>

```

## sse\_server.js

```

var express = require("express");
var app = express();

var connectedClients = [];

app.use(express.static(__dirname));

var port = process.env.PORT || 8080 ;

app.listen(port, function() {
    console.log("Listening on " + port);
});

app.use(express.json());
app.use(express.urlencoded());

app.get('/sse/:timestamp', function (req, res) {
    var pingSent = parseInt(req.param('timestamp'));

    req.socket.setTimeout(Infinity);
    res.writeHead(200, {
        'content-type': 'text/event-stream',
        'connection': 'keep-alive',
        'access-control-allow-origin': '*'
    });

    var reply = {
        "pingSent": pingSent,
        "pingReceived": Date.now()
    };

    res.write("event: " + "pingReply" + "\n");
    res.write("data: " + JSON.stringify(reply) + "\n\n");
});

```

## sse\_client.html

```

<!doctype html>
<html>
<head>
    <script type="text/javascript" src="../jquery-2.1.0.min.js"></script>
    <script type="text/javascript">
$(document).ready(function() {
    $("button").click(function() {

```

```

    var source = new EventSource('http://' + document.domain + ':8080/sse/'
+ Date.now());

    source.addEventListener("pingReply", function(e) {
        var msg = JSON.parse(e.data);
        msg["replyReceived"] = Date.now();
        $("#output").html("Ping sent: \t\t" + msg["pingSent"] + "\nPing
received: \t\t" + msg["pingReceived"] + "\nReply received: \t" +
msg["replyReceived"] + "\nLatency: \t\t" + (msg["replyReceived"]-
msg["pingSent"]) + "ms");
        source.close();
    }, false);

});
});
</script>
<body>
<div>
<button>Ping!</button>
<p id="output" style="white-space: pre; font-family: monospace"></p>
</div>
</body>
</html>

```

## Ajax\_sever.js

```

var express = require("express");
var app = express();

var connectedClients = [];

app.use(express.static(__dirname));
app.use(express.json());
app.use(express.urlencoded());

var port = process.env.PORT || 8080 ;

app.listen(port, function() {
    console.log("Listening on " + port);
});

app.post('/message', function(req, res) {
    console.log(req.body);

    var reply = {
        "pingSent": parseInt(req.body.pingSent),
        "pingReceived": Date.now()
    };

    res.setHeader('Content-Type', 'application/json');
    res.end(JSON.stringify(reply));
});

```

## Ajax\_client.html

```

<!doctype html>
<html>
<head>
    <script type="text/javascript" src="../jquery-2.1.0.min.js"></script>
    <script type="text/javascript">
$(document).ready(function() {
    $("button").click(function() {

```

```

$.post('/message', {"pingSent": Date.now()}, function(data) {
  var msg = data;
  msg["replyReceived"] = Date.now();
  $("#output").html("Ping sent: \t\t" + msg["pingSent"] + "\nPing
received: \t\t" + msg["pingReceived"] + "\nReply received: \t" +
msg["replyReceived"] + "\nLatency: \t\t" + (msg["replyReceived"]-
msg["pingSent"])) + "ms");
  });
});
});
</script>
<body>
  <div>
    <button>Ping!</button>
    <p id="output" style="white-space: pre; font-family: monospace"></p>
  </div>
</body>
</html>

```

## Appendix B (Pilot study measurement data)

	WebSocket	SSE	Ajax
Local machine 1	4	14	24
Local machine 2	1	13	6
Local machine 3	1	317	5
Local machine 4	1	11	5
Local machine 5	2	313	4
Local Machine Average	1,8	133,6	8,8
Wifi LAN 1	6	219	8
Wifi LAN 2	5	217	6
Wifi LAN 3	5	219	8
Wifi LAN 4	2	220	7
Wifi LAN 5	3	207	18
WIFI LAN Average	4,2	216,4	9,4
3G 1	40	125	147
3G 2	37	134	101
3G 3	81	184	166
3G 4	34	123	184
3G 5	83	149	143
3G Average	55	143	148,2

## Appendix C – WebSocket application source code

### server.js

```
var WebSocketServer = require('ws').Server,
    wss = new WebSocketServer({port: 1234});

var connectedClients = [];

var express = require('express');
var app = express();

app.use(express.static(__dirname));

var port = process.env.PORT || 8080;
app.listen(port);

wss.on('connection', function(ws) {

    ws.on('close', function(message) {
        var index = connectedClients.indexOf(ws);
        if(index > -1 ) {
            connectedClients.splice(index, 1);
            var broadcastObject = {
                "type": "userDisconnected",
                "username": ws.username
            }

            broadcast(broadcastObject);
        }
    });

    ws.on('message', function(message) {
        var messageObject = JSON.parse(message);

        if(messageObject["type"] == "firstConnection") {
            var nameTaken = false;

            for(var i = 0; i < connectedClients.length; i++) {
                if(connectedClients[i].username == messageObject["username"]) {
                    nameTaken = true;
                    ws.send(JSON.stringify({
                        "type": "usernameTaken",
                        "username": messageObject["username"]
                    }));
                    break;
                }
            }

            if(!nameTaken) {
                ws.username = messageObject["username"];
                var userlist = [];

                connectedClients.forEach(function(e) {
                    userlist.push(e.username);
                });

                ws.send(JSON.stringify({
                    "type": "connectionSuccessful",
                    "userlist": userlist
```

```

    ));

    connectedClients.push(ws);

    broadcast({
      "type": "userConnected",
      "username": ws.username
    });
  }
}

if(messageObject["type"] == "chatMessage") {
  var reply = messageObject;
  reply["serverSent"] = Date.now();
  reply["author"] = ws.username;
  broadcast(reply);
}

if(messageObject["type"] == "ping") {
  ws.send(JSON.stringify({
    "type": "pingReply"
  }));
}
});
});

function broadcast(message) {
  for(var i = 0; i < connectedClients.length; i++) {
    connectedClients[i].send(JSON.stringify(message));
  }
}

```

## client.js

```

/* --- GUI stuff --- */

window.onload = function() {

  $("#intro form").submit(function (e) {
    e.preventDefault();
    if(!$("#username").val()) {
      $(this).append("<p>Please enter a username!</p>");
    } else {
      setupWs($("#username").val());
    }
  });

  $("#controls form").submit(function (e) {
    e.preventDefault();
    handleInputBox();
  });

  $("#controls #pingButton").click(handlePing);

  message.value = "";

  // When the user scrolls, disable autoscrolling (moving text window down)
  $("#output").scroll(function() {
    window.autoscroll = false;

    // If the user scrolls all the way to the bottom, enable autoscrolling
    var output = $("#output")[0];
  });

```

```

        if( output.scrollTop == (output.scrollHeight - output.offsetHeight)) {
            window.autoscroll = true;
        }

    });
};

function handlePing() {
    systemMessage("Pinging server...");
    var messageObject = {
        "type": "ping"
    }

    window.latestPing = Date.now();

    ws.send(JSON.stringify(messageObject));
}

function handleInputBox() {
    var box = document.getElementById("message");
    var message = box.value;

    if(message != '') {
        var messageObject = {
            "type": "chatMessage",
            "clientSent": Date.now(),
            "content": message
        }

        ws.send(JSON.stringify(messageObject));
        box.value = "";
        handleScroll(true);
    }
}

function populateUserlist(list) {
    list.forEach(function (e) {
        $("#connectedUsers").append('<li>' + e + '</li>');
    });
}

function getTimestamp() {
    var now = new Date();
    var timestamp = ("0" + now.getHours()).slice(-2) + ":" + ("0" +
now.getHours()).slice(-2) + ":" + ("0" + now.getSeconds()).slice(-2);
    return timestamp;
}

function handleScroll(force) {
    if(window.autoscroll != false || force == true) {
        var output = $("#output")[0];
        $("#output").scrollTop(output.scrollHeight - output.offsetHeight);
    }
}

function systemMessage(content) {
    $("#output").append('<div
class="systemMessage"><time>'+getTimestamp()+'</time><span
class="username">system</span><p class="text">'+ content + '</p></div>');
    handleScroll();
}

```



```

function userConnected(username) {
    $("#connectedUsers").append('<li>' + username + '</li>');
    systemMessage(username + " has connected.");
}

function userDisconnected(username) {
    $("#connectedUsers li").filter(function(index) { return $(this).text()
=== username; }).remove();
    systemMessage(username + " has disconnected.");
}

function chatMessage(username, content) {
    $("#output").append('<div
class="message"><time>' + getTimestamp() + '</time><span
class="username">' + username + '</span><blockquote
class="text">' + content + '</blockquote></div>');
    handleScroll();
}

function usernameTaken() {
    $("#intro form").append('<p>Username in use! Try a different one.</p>');
    window.ws.close();
}

/* --- Handling the WebSocket connection --- */

function setupWs(username) {
    window.ws = new WebSocket("ws://localhost:1234");

    ws.onopen = function(event) {
        ws.send(JSON.stringify({
            "type": "firstConnection",
            "username": username
        }));
    };

    ws.onmessage = function(event) {
        var messageObject = JSON.parse(event.data);

        if(messageObject["type"] == "chatMessage") {
            chatMessage(messageObject["author"], messageObject["content"]);
            messageObject["clientReceived"] = Date.now();
            messageObject["roundTrip"] = (messageObject["clientReceived"] -
messageObject["clientSent"]);
            console.log(messageObject);
        }

        if(messageObject["type"] == "userConnected") {
            userConnected(messageObject["username"]);
        }

        if(messageObject["type"] == "userDisconnected") {
            userDisconnected(messageObject["username"]);
        }

        if(messageObject["type"] == "usernameTaken") {
            usernameTaken();
        }

        if(messageObject["type"] == "connectionSuccessful") {

```

```
        populateUserlist(messageObject["userlist"]);
        document.getElementById("intro").style.display="none";
    }

    if(messageObject["type"] == "pingReply") {
        var diff = (Date.now()-window.latestPing);
        systemMessage("Latency: " + diff + "ms");
    }
};
}
```

## Appendix D – SSE client application source code

### server.js

```
var express = require("express");
var app = express();

var connectedClients = [];

app.use(express.static(__dirname));

var port = process.env.PORT || 1234;

app.listen(port, function() {
  console.log("Listening on " + port);
});

app.use(express.json());
app.use(express.urlencoded());

app.get('/es/:username', function (req, res) {
  console.log("NEW USER!!!");
  req.socket.setTimeout(Infinity);
  res.writeHead(200, {
    'content-type': 'text/event-stream',
    'connection': 'keep-alive',
    'access-control-allow-origin': '*'
  });

  // Append username fetched from the Express param :username onto the
  response object for the current connection
  res.username = req.param("username");

  send("connections successful", "", res);

  connectedClients.push(res);

  broadcast({
    "type": "userConnected",
    "username": res.username
  });
});

app.get('/userlist', function (req, res){
  res.json(getUserList());
});

app.post('/message', function(req, res){
  if(req.body.type == "chatMessage") {
    var reply = req.body;
    reply["serverSent"] = Date.now();

    broadcast(reply);
  } else if(req.body.type == "ping") {
    sendTo(req.body.author, "pingReply", "");
  }
});

function broadcast(message) {
  for(var i = 0; i < connectedClients.length; i++) {
```

```

        send("message", JSON.stringify(message), connectedClients[i]);
    }
}

function sendTo(username, ev, content) {
    for(var i = 0; i < connectedClients.length; i++) {
        if(connectedClients[i].username == username) {
            send(ev, content, connectedClients[i]);
        }
    }
}

function send(ev, content, responseObject) {
    responseObject.write("event: " + ev + "\n");
    responseObject.write("data: " + content + "\n\n");
}

function getUserList() {
    var ul = [];

    for(var i = 0; i < connectedClients.length; i++) {
        ul.push(connectedClients[i].username);
    }

    return ul;
}

```

## client.js

```

$(document).ready(function() {

    $("#intro form").submit(function (e) {

        e.preventDefault();
        if(!$("#username").val()) {
            $(this).append("<p>Please enter a username!</p>");
        } else {

            window.username = $("#username").val();

            $.get('/userlist', function(data){
                window.userlist = data;
                if(data.indexOf(window.username) == -1) {
                    setupSSE(window.username);
                } else {
                    usernameTaken();
                }
            });
        }
    });

    $("#controls form").submit(function (e) {
        e.preventDefault();
        handleInputBox();
    });

    $("#controls #pingButton").click(handlePing);

    message.value = "";

```

```

// When the user scrolls, disable autoscrolling (moving text window down)
$("#output").scroll(function() {
    window.autoscroll = false;

    // If the user scrolls all the way to the bottom, enable autoscrolling
    var output = $("#output")[0];
    if( output.scrollTop == (output.scrollHeight - output.offsetHeight)) {
        window.autoscroll = true;
    }

});

});

/* SSE Stuff */

function handleInputBox() {
    var box = document.getElementById("message");
    var message = box.value;

    if(message != '') {

        var messageObject = {
            "type": "chatMessage",
            "clientSent": Date.now(),
            "content": message,
            "author": window.username
        }

        $.Ajax({
            type: "POST",
            url: '/message',
            data: messageObject,
            success: function(one, two, three) { console.log(one);
console.log(two); console.log(three); },
            dataType: "json"
        });

        console.log("post");
        box.value = "";
        handleScroll(true);
    }
}

function handlePing() {
    systemMessage("Pinging server...");
    window.latestPing = Date.now();
    $.post('/message', {"type": "ping", "author": window.username});
}

function setupSSE(username) {
    console.log("sse ev");

    var source = new
EventSource('http://' + document.domain + ':1234/es/' + username);

    source.addEventListener("open", function(e) {
        $("#intro").hide();
        populateUserlist(window.userlist);
    }, false);
}

```

```

source.addEventListener("pingReply", function(e) {
    var diff = (Date.now()-window.latestPing);
    systemMessage("Latency: " + diff + "ms");
}, false);

source.addEventListener("message", function(e) {
    var messageObject = JSON.parse(e.data);

    if(messageObject["type"] == "chatMessage") {
        chatMessage(messageObject["author"], messageObject["content"]);
        messageObject["clientReceived"] = Date.now();
        messageObject["roundTrip"] = (messageObject["clientReceived"]-
messageObject["clientSent"]);
        console.log(messageObject);
    }

    if(messageObject["type"] == "userConnected") {
        userConnected(messageObject["username"]);
    }
}, false);

}

/* Generic GUI Stuff */

function getTimestamp() {
    var now = new Date();
    var timestamp = ("0" + now.getHours()).slice(-2) + ":" + ("0" +
now.getHours()).slice(-2) + ":" + ("0" + now.getSeconds()).slice(-2);
    return timestamp;
}

function handleScroll(force) {
    if(window.autoscroll != false || force == true) {
        var output = $("#output")[0];
        $("#output").scrollTop(output.scrollHeight - output.offsetHeight);
    }
}

function systemMessage(content) {
    $("#output").append('<div
class="systemMessage"><time>'+getTimestamp()+'</time><span
class="username">system</span><p class="text">'+ content + '</p></div>');
    handleScroll();
}

function userConnected(username) {
    $("#connectedUsers").append('<li>' + username + '</li>');
    systemMessage(username + " has connected.");
}

function userDisconnected(username) {
    $("#connectedUsers li").filter(function(index) { return $(this).text()
=== username; }).remove();
    systemMessage(username + " has disconnected.");
}

function chatMessage(username, content) {

```

```

    $("#output").append('<div
class="message"><time>'+getTimestamp()+'</time><span
class="username">'+username+'</span><blockquote
class="text">'+content+'</blockquote></div>');
    handleScroll();
}

function usernameTaken() {
    $("#intro form").append('<p>Username in use! Try a different one.</p>');
}

function populateUserlist(list) {
    list.forEach(function (e) {
        $("#connectedUsers").append('<li>' + e + '</li>');
    });
}

```

## Appendix E – Ajax application source code

### server.js

```
var express = require('express');
var fs = require('fs');
var app = express();

var userlist = [];

app.use(express.static(__dirname));
app.use(express.json());
app.use(express.urlencoded());

var port = process.env.PORT || 8080;
var messageLimit = 10;
var messages = [];

app.listen(port, function() {
  console.log("Listening on " + port);
});

app.get('/read', function(req, res) {
  fs.readFile("files/boop.txt", function(err, data) {
    if(err) {
      console.log(err);
    } else {
      res.send(data);
    }
  });
});

app.get('/userlist', function(req, res) {
  res.json(userlist);
});

app.post('/message', function(req, res) {
  var message = req.body;
  message["serverReceived"] = Date.now();

  trimArray(messages, messageLimit);

  messages.push(message);
});

app.post('/userdisconnect', function(req, res) {
  userlist.pop(userlist.indexOf(req.body.username));
});

app.get('/fetchmessages/:username', function(req, res) {
  var username = req.param("username");
  res.json(messages);
});

app.post('/connect', function(req, res) {
  var username = req.body.username;
  if(userlist.indexOf(username) == -1) {
    userlist.push(username);
    res.end("ok");
  } else {
```



```

        res.end("usernameinuse");
    }
});

app.get('/ping', function(req, res) {
    var reply = req.body;
    reply["serverReceived"] = Date.now();
    res.json(reply);
});

function trimArray(arr, len) {
    if(arr.length > len) {
        console.log("Overflow!");
        while(arr.length > len) {
            arr.shift();
        }
    }
}

```

## client.js

```

$(document).ready(function() {

    window.messages = [];
    window.userlist = [];

    $("#intro form").submit(function (e) {

        e.preventDefault();
        if(!$("#username").val()) {
            $("#intro form").append("<p>Please enter a username!</p>");
        } else {
            setupAjax($("#username").val());
        }
    });

    $("#controls form").submit(function (e) {
        e.preventDefault();
        handleInputBox();
    });

    $("#controls #pingButton").click(handlePing);

    message.value = "";

    // When the user scrolls, disable autoscrolling (moving text window down)
    $("#output").scroll(function() {
        window.autoscroll = false;

        // If the user scrolls all the way to the bottom, enable autoscrolling
        var output = $("#output")[0];
        if( output.scrollTop == (output.scrollHeight - output.offsetHeight)) {
            window.autoscroll = true;
        }

    });

});

function populateUserList(list) {
    list.forEach(function (e) {
        $("#connectedUsers").append('<li>' + e + '</li>');
    });
}

```

```

    });
}

function setupAjax(username) {
    window.username = username;
    $.post('/connect', {"username": username}, function(data) {
        if(data == "ok") {
            $("#intro").hide();
            $.get('/userlist', function(data) {
                window.userlist = data;
                populateUserList(data);
            });
            window.onbeforeunload = function() {
                $.post('/userdisconnect', {"username": username});
            };
            setInterval(primaryLoop, 200);
        } else {
            usernameTaken();
        }
    });
}

function handleInputBox() {
    var message = $("#message").val();

    if(message != '') {

        var messageObject = {
            "type": "chatMessage",
            "clientSent": Date.now(),
            "content": message,
            "author": window.username
        }

        $.post('/message', messageObject, function(data) {});

        $("#message").val("");
        handleScroll(true);
    }
}

function primaryLoop() {
    $.get('/fetchmessages/'+window.username, function(data) {
        if(data.length > 0) {
            sortMessages(data);
        }
    });

    $.get('/userlist', function(data) {
        for(var i = 0; i < data.length; i++) {
            if(window.userlist.indexOf(data[i]) == -1) {
                userConnected(data[i]);
            }
        }

        for(var i = 0; i < window.userlist.length; i++) {
            if(data.indexOf(window.userlist[i]) == -1) {
                userDisconnected(window.userlist[i]);
            }
        }
    });
}

```

```

}

function sortMessages(fetchedMessages) {
    // There is no local list. Fetch foreign list, make local list and parse
    messages.
    if(window.messages.length == 0) {
        window.messages = fetchedMessages;
        parseMessages(fetchedMessages);
        return;
    }

    // Local list exists but is not identical to foreign list
    // Append missing elements to local list, parse missing messages
    if(window.messages[(window.messages.length-1)]["clientSent"] !=
    fetchedMessages[(fetchedMessages.length-1)]["clientSent"]) {

        var difference = getDifference(window.messages, fetchedMessages);
        window.messages = window.messages.concat(difference);
        parseMessages(difference);
        return;
    }
}

function getDifference(arr1, arr2) {
    var result = [];
    for(var i = 0; i < arr2.length; i++) {
        if(inArray(arr1, arr2[i]) == false) {
            result.push(arr2[i]);
        }
    }

    return result;
};

function inArray(arr, obj) {
    for(var i = 0; i < arr.length; i++) {
        if(arr[i]["clientSent"] == obj["clientSent"]) {
            return true;
        }
    }

    return false;
}

function parseMessages(data) {
    for(var i = 0; i < data.length; i++) {
        parseMessage(data[i]);
    }
}

function parseMessage(msg) {
    if(msg["type"] == "chatMessage") {
        chatMessage(parseInt(msg["clientSent"]), msg["author"],
msg["content"]);
        msg["clientReceived"] = Date.now();
        msg["roundTrip"] = (msg["clientReceived"]-msg["clientSent"]);
        console.log(msg);
    }

    if(msg["type"] == "userConnected") {
        userConnected(msg["username"]);
    }
}

```

```

    }
}

function handlePing() {
    var now = Date.now();
    systemMessage("Pinging server...");
    $.get("/ping", function(data){
        var later = Date.now();
        systemMessage("Latency: " + (later-now) + "ms");
    });
}

/* Generic GUI Stuff */

function getTimestamp() {
    var now = new Date();
    var timestamp = ("0" + now.getHours()).slice(-2) + ":" + ("0" +
now.getHours()).slice(-2) + ":" + ("0" + now.getSeconds()).slice(-2);
    return timestamp;
}

function formatTimestamp(timestamp) {
    var now = new Date(timestamp);
    var timestamp = ("0" + now.getHours()).slice(-2) + ":" + ("0" +
now.getHours()).slice(-2) + ":" + ("0" + now.getSeconds()).slice(-2);
    return timestamp;
}

function handleScroll(force) {
    if(window.autoscroll !== false || force == true) {
        var output = $("#output")[0];
        $("#output").scrollTop(output.scrollHeight - output.offsetHeight);
    }
}

function systemMessage(content) {
    $("#output").append('<div
class="systemMessage"><time>'+getTimestamp()+'</time><span
class="username">system</span><p class="text">'+ content + '</p></div>');
    handleScroll();
}

function userConnected(username) {
    window.userlist.push(username);
    $("#connectedUsers").append('<li>' + username + '</li>');
    systemMessage(username + " has connected.");
}

function userDisconnected(username) {
    $("#connectedUsers li").filter(function(index) { return $(this).text()
=== username; }).remove();
    window.userlist.pop(window.userlist.indexOf(username));
    systemMessage(username + " has disconnected.");
}

function chatMessage(timestamp, username, content) {
    $("#output").append('<div
class="message"><time>'+formatTimestamp(timestamp)+'</time><span
class="username">'+username+'</span><blockquote
class="text">'+content+'</blockquote></div>');
    handleScroll();
}

```

```
}  
  
function usernameTaken() {  
    $("#intro form").append('<p>Username in use! Try a different one.</p>');  
}
```

## Appendix F – Measurement script

### testing.js

```
$(document).on("connectionEstablished", function() {
    $("#controls form").append('<button id="testing">Test!</button>');
    $("#controls form").append('<button id="printMessages">Print local
messages</button>');

    $("#controls #testing").click(function() { runTest(100)});
    $("#controls #printMessages").click(function() { printMessages()});
});

function runTest(amt) {
    var i = 0;

    var interval = setInterval(function() {
        $("#controls form #message").val(randomString());
        $("#controls form").submit();
        i++;
        if(i >= amt) {
            clearInterval(interval);
        }
    }, 200);
}

function randomString() {
    var words = ["hello", "world", "foo", "bar", "baz", "one", "two",
"three", "four", "five", "six", "seven", "eight", "nine"];
    var length = getRandomInt(5, 25);
    var result = "";

    for(var i = 0; i < length; i++) {
        result += words[getRandomInt(0, words.length-1)] + " ";
    }

    return result.trim();
}

function getRandomInt(min, max) {
    return Math.floor(Math.random() * (max - min + 1)) + min;
}

function printMessages() {
    var $table = $('<table><tr><th>Author</th><th>Content
Size</th><th>ClientSent</th><th>ServerSent</th><th>ClientReceived</th><th>R
oundTrip</th></tr></table>');
    for(var i = 0; i < messages.length; i++) {
        $row = $('<tr>');
        $row.append('<td>' + messages[i]['author'] + '</td>');
        $row.append('<td>' + messages[i]['content'].length + '</td>');
        $row.append('<td>' + messages[i]['clientSent'] + '</td>');
        $row.append('<td>' + messages[i]['serverSent'] + '</td>');
        $row.append('<td>' + messages[i]['clientReceived'] + '</td>');
        $row.append('<td>' + messages[i]['roundTrip'] + '</td>');

        $table.append($row);
    }
}
```

```

var $result = $('<div>')
.append($table)
.css({
    "position": "absolute",
    "top": 0,
    "left": 0,
    "height": "500px",
    "right": 0,
    "background": "white",
    "z-index": 1000,
    "color": "black",
    "overflow": "auto",
    "font-family": "monospace"
})
.dblclick(function() { $(this).remove(); });

$(document.body).append($result);
}

```

## Appendix G – Local Machine Measurement Data

Ajax	SSE	WebSocket
39	22	4
40	5	3
40	7	1
40	9	4
45	7	6
41	10	2
42	6	5
41	8	4
47	8	4
39	8	4
38	7	4
38	8	8
36	7	8
38	7	3
34	7	6
33	6	5
34	6	4
39	6	5
41	6	5
36	7	5
37	5	2
37	6	2
37	8	2
34	6	2
35	4	3
36	4	4
39	4	4
38	6	5
38	4	5
39	5	5
38	7	5
42	8	3
42	7	5
44	7	6
45	8	6
38	4	5
41	8	3
39	7	3
36	6	3
36	9	2
35	9	1
35	5	2
34	8	2
38	11	3
35	4	1
37	8	3
41	8	2
39	4	6
38	8	6
37	9	5
37	4	5



37	7	5
38	10	3
37	8	8
36	6	8
37	5	6
38	4	6
38	5	2
37	9	6
40	5	3
44	9	3
39	8	6
41	5	3
39	8	4
42	4	3
40	7	3
41	10	2
38	8	3
41	6	3
42	8	3
41	9	7
41	7	4
45	4	2
49	4	3
53	7	3
46	7	3
46	9	1
45	8	4
46	6	4
46	8	4
43	22	20
43	4	6
43	9	6
46	5	7
47	4	6
44	7	6
47	4	6
46	8	3
44	8	3
46	9	3
43	7	3
44	8	3
48	8	6
44	7	6
48	24	7
48	4	4
44	6	3
52	5	3
53	11	2
53	11	2

# Appendix H – LAN Measurement Data

Measurement 1

Ajax	SSE	WebSocket
56	232	9
47	34	7
44	237	8
43	37	5
45	45	8
45	56	7
45	64	6
45	73	6
45	84	5
45	94	5
46	96	5
46	96	6
44	95	7
44	94	6
47	93	6
43	93	8
42	92	6
42	91	7
41	93	5
43	93	5
44	93	5
43	92	21
42	91	5
54	90	6
41	90	9
41	91	8
43	91	7
42	90	9
41	90	6
42	89	5
43	89	6
42	89	8
44	96	6
76	88	8
111	88	9
80	92	6
110	91	6
43	90	7
57	90	7
42	90	7
42	97	7
42	99	7
42	99	9
38	99	7
43	97	7
43	99	7
42	97	7
377	97	7
242	97	7

Measurement 2

Ajax	SSE	WebSocket
176	218	4
174	20	6
172	18	13
172	139	8
175	188	5
177	189	5
175	193	5
173	195	5
172	198	5
175	196	8
174	196	7
173	192	7
172	194	9
172	192	8
173	203	6
170	207	6
171	208	5
177	208	5
175	211	8
176	11	5
175	12	5
174	13	5
173	13	5
173	13	6
174	14	6
176	15	5
173	19	6
174	22	6
173	24	5
177	24	6
172	25	6
174	25	6
177	26	4
174	34	15
179	28	8
170	32	7
166	32	6
166	32	6
170	33	6
171	34	4
172	44	7
175	45	8
172	47	6
176	48	6
176	48	7
195	49	6
160	50	6
155	50	27
156	51	6

Measurement 3

Ajax	SSE	WebSocket
28	253	9
30	278	8
32	281	9
34	81	7
32	82	6
30	82	6
36	82	6
28	83	5
27	84	5
29	86	5
27	86	5
28	86	4
28	90	8
27	91	5
28	93	8
28	93	3
27	95	5
27	96	5
29	97	3
28	97	5
29	108	5
27	109	5
27	108	6
28	114	6
27	118	6
26	122	5
26	127	6
26	132	6
29	136	8
27	137	6
31	138	4
27	139	5
26	140	16
27	154	9
35	148	8
28	149	7
27	152	8
28	164	7
28	167	6
27	177	6
28	182	6
27	181	6
27	180	7
26	176	6
25	179	7
26	180	6
29	181	6
26	181	6
26	177	6

42	97	10
41	98	10
40	99	9
48	108	8
40	105	7
39	105	7
39	105	7
40	102	7
40	104	7
30	104	6
28	104	9
27	104	7
27	104	7
26	104	7
30	104	6
25	104	7
26	104	8
26	106	8
28	104	4
25	106	9
25	113	8
25	115	7
25	125	10
55	134	7
25	145	7
25	153	7
25	160	7
27	163	7
28	162	7
28	171	9
46	172	7
29	182	11
28	189	7
29	189	7
29	178	7
30	179	7
28	178	7
34	178	7
28	187	7
28	190	7
29	189	7
27	179	7
28	189	7
29	189	7
28	176	7
29	177	7
29	177	11
30	177	10
30	176	9
29	176	7
30	176	7

154	51	6
155	52	6
155	53	6
155	54	6
157	54	9
163	55	7
161	61	4
163	63	7
163	64	9
163	64	9
163	67	8
164	69	5
162	71	7
163	78	7
162	83	9
162	84	7
163	86	7
162	97	7
166	101	7
167	106	7
161	111	7
165	113	8
177	120	7
164	110	7
163	116	7
163	121	7
164	124	7
163	129	7
167	134	7
166	139	8
166	143	7
164	163	7
161	164	7
164	164	7
164	167	7
162	181	7
162	181	3
163	181	6
163	187	7
164	181	5
163	181	5
163	185	5
164	182	8
163	182	7
162	182	7
161	180	7
163	184	7
177	190	7
162	181	7
163	183	12
164	183	8

26	181	6
25	176	6
25	181	6
26	183	7
26	181	9
28	181	8
28	180	6
29	182	7
38	182	7
25	178	8
24	184	10
23	181	9
25	180	11
23	184	9
24	181	7
24	177	10
24	179	7
23	183	7
24	183	7
23	180	7
28	179	7
25	179	510
26	179	312
23	186	17
23	194	26
25	180	5
25	181	3
24	179	4
24	183	4
23	185	4
113	183	4
111	183	7
105	182	6
133	184	14
26	184	14
68	198	8
23	182	7
23	182	12
22	180	7
22	180	8
23	181	7
23	182	7
23	191	7
22	162	7
23	166	7
23	169	7
24	171	9
24	174	7
24	177	10
22	180	7
24	181	7

# Appendix I – 4G Measurement Data

## Measurement 1

2014-05-08 14:00

Ajax	SSE	WebSocket
103	331	53
76	136	74
76	169	80
75	251	98
75	309	73
73	110	68
75	153	70
75	190	68
71	249	67
73	828	79
71	628	63
71	432	62
67	234	75
68	279	62
69	328	63
86	130	63
88	628	63
87	430	63
85	232	68
89	279	81
83	329	65
84	132	71
83	195	58
83	247	66
84	307	57
81	109	99
85	169	82
82	224	81
79	272	94
84	75	85
80	123	86
101	184	61
78	243	63
63	301	72
72	101	76
76	163	76
269	219	85
69	270	73
68	320	70
91	121	82
99	183	69
97	236	76
74	287	77
79	336	75
98	138	68
99	196	81
91	249	65

## Measurement 2

2014-05-16 11:00

Ajax	SSE	WebSocket
62	304	206
73	105	38
67	131	46
69	172	42
69	211	56
75	254	40
89	56	43
66	71	39
65	104	33
73	135	44
63	175	34
70	219	41
70	262	44
68	301	46
81	102	62
319	140	48
273	177	40
73	220	50
71	249	45
77	277	43
82	78	40
76	104	51
64	137	42
77	186	60
62	224	43
72	255	42
80	283	56
75	86	61
57	153	36
70	190	54
73	332	39
78	358	54
74	410	48
85	213	50
79	254	45
83	284	42
77	86	50
74	116	60
80	151	46
78	183	62
107	230	45
76	260	47
85	62	41
80	81	42
77	111	40
78	151	37
83	184	41

## Measurement 3

2015-05-15 22:03

Ajax	SSE	WebSocket
121	362	72
151	166	205
123	241	67
121	281	67
118	319	111
150	119	111
119	206	73
117	284	139
115	315	73
120	117	62
116	200	74
117	229	62
119	268	75
213	307	69
108	107	61
150	189	66
114	264	68
164	302	71
111	102	59
157	140	70
104	218	57
149	258	60
105	305	132
159	336	53
104	137	106
118	465	51
158	267	97
100	294	96
145	333	104
127	134	57
142	170	91
124	259	100
134	300	54
106	330	96
139	130	97
142	214	41
123	294	97
102	327	97
131	128	44
131	328	86
177	409	86
161	209	84
147	518	84
156	603	77
126	404	37
215	205	89
136	211	38

67	312	64
99	112	64
71	150	43
97	209	63
86	267	78
87	307	52
93	108	76
82	166	64
82	206	63
68	263	67
84	303	60
85	105	60
97	162	60
85	218	59
94	294	73
84	339	60
79	141	92
61	199	60
84	255	58
84	315	130
81	117	56
80	170	56
686	233	68
678	279	62
479	331	85
280	134	89
80	196	52
81	245	52
79	304	78
78	107	141
81	164	51
80	225	53
85	274	127
79	324	58
74	127	61
73	182	70
73	245	49
77	305	50
77	106	57
75	139	49
72	203	49
71	255	47
74	311	70
99	112	48
116	178	74
68	252	54
69	307	66
72	109	66
105	151	64
80	211	73
69	412	44
80	468	79
84	270	84

80	218	47
89	248	51
82	286	40
81	88	38
86	127	40
108	164	48
89	203	51
86	224	44
85	240	56
79	273	40
90	74	46
94	104	36
72	135	42
71	175	45
71	215	48
68	246	50
68	272	40
71	75	47
72	102	39
73	133	46
72	168	164
72	208	115
69	269	118
73	300	39
62	102	102
61	135	154
60	167	33
59	209	52
55	264	50
76	300	106
68	102	43
69	124	61
61	159	44
66	200	49
73	233	49
65	258	46
57	60	50
64	88	37
69	114	41
72	147	55
67	186	59
66	218	48
54	255	44
70	58	56
63	78	46
63	104	55
254	135	33
55	175	49
72	215	42
70	245	41
57	273	58
53	75	57
64	100	49

214	361	82
133	453	36
151	255	80
122	289	86
131	327	74
122	127	79
151	167	78
118	250	90
116	333	72
115	134	75
119	212	85
115	253	72
114	285	71
123	86	89
113	120	77
111	229	89
111	280	80
126	81	27
107	172	80
104	236	80
151	277	72
105	315	94
144	116	70
106	154	75
148	231	62
102	280	63
127	81	70
165	154	69
109	228	62
145	324	60
128	391	60
128	193	58
128	252	65
113	291	58
103	331	57
132	131	92
122	214	55
133	294	62
100	374	149
131	176	98
100	246	155
138	287	50
99	88	58
134	122	96
118	209	49
103	320	104
125	401	47
131	202	185
154	215	39
125	305	84
133	107	82
213	179	92
122	215	36