Real-Time Web with WebSocket

Comparing Real-Time Transporting Technologies for the Web

Øyvind Raasholm Tangen
Master's Thesis
Department of Informatics
University of Oslo
oyvindrt@ifi.uio.no / oyvindrtangen@gmail.com

May 13th, 2015

Abstract

Background: A real-time web application is a website that dynamically updates itself as soon as new content is available. This functionality is made possible by different technologies and network protocols commonly referred to as *transports* in this thesis.

Aim: This thesis compares the real-time transports WebSocket, Server-Sent Events and HTTP Long Polling in terms of performance and programmer friendliness.

Method: Two distinct test scenarios created to performance test the transports. The first is a unidirectional messaging system with a server broadcasting messages to connected clients. The second is a bidirectional chat system where a set of clients send and receive messages to and from a chat server.

Results: The thesis suggests a two-sided picture of the transports. WebSocket outperforms both Server-Sent Events and HTTP Long Polling under moderate load levels. However, as soon as the server is CPU constrained, we see that both WebSocket and Server-Sent Events had unexpected increases in response times and memory consumption.

Conclusion: Under moderate server CPU load levels, WebSocket is recommended over Server-Sent Events and HTTP Long Polling, both in terms of performance and programmer friendliness. However, under extreme levels of server CPU load, HTTP Long Polling is the preferred transport.

Contents

Abstract 3
Contents 4
List of Figures 8
List of Examples 9
List of Tables 9
Preface 10
Acknowledgements 11
Chapter 1: Introduction 13
1.1 Motivation 13
1.2 Research Questions 14
1.3 Thesis Outline 14
Chapter 2: Background 15
2.1 Related Academic Work 15
2.2 TCP 16
2.3 The Web and HTTP 16
2.3.1 The World Wide Web 16
2.3.2 HTTP 17
2.3.3 HTTP Methods 17
2.3.4 HTTP Header Fields and State 17
2.4 Modern Web 19
2.4.1 Ajax 19
2.5 Real-Time HTTP 19
2.5.1 HTTP Polling 20
2.5.2 Comet 20
2.5.3 Why Comet and HTTP is Unsatisfactory 21
2.5.4 HTML5 22
2.6 Server-Sent Events 23
2.6.1 EventSource API 23
2.6.2 Event Stream Protocol 23
2.6.3 Server-Sent Events Problems 24
2.7 WebSocket 25
2.7.1 The WebSocket API 25
2.7.2 The WebSocket Protocol 26
2.7.3 WebSocket and HTTP 28
2.7.4 WebSocket vs. Server-Sent Events and HTTP 29

2.8 Web Platforms & Node.js 30
2.8.1 Web Development 30
2.8.2 Node.js 31
2.9 Performance Testing 33
2.9.1 Introduction to Performance Testing 33
2.9.2 Response Times 34
2.10 Test Expectations 34
Chapter 3: Methodology 37
3.1 Test Scenarios 37
3.1.1 Test Scenario 1 37
3.1.2 Test Scenario 2 38
3.2 Test Data 39
3.2.1 Two Points of View 39
3.2.2 Collection through Three Test Phases 39
3.2.3 Collection 40
3.2.4 Number of Test Runs 40
3.3 Testing environment 40
3.3.1 Hardware 40
3.3.2 Programming environment 41
3.3.3 Command Line Clients 43
3.4 Test Configurations 43
3.4.1 Maximum Number of Clients 43
3.4.2 Parameters Specific to the First Scenario 44
3.4.3 Parameters Specific to the Second Scenario 44
3.5 Detailed Information Flow 45
3.5.1 Test Scenario 1 46
3.5.2 Test Scenario 2 47
3.6 Development 49
3.6.1 Common between Scenarios 49
3.6.2 Software Versions 50
3.6.3 Scenario 1 Specific Implementation Details 51
3.6.4 Scenario 2 Specific Implementation Details 52
3.7 Limitations 54
3.7.1 Performance over Longer Periods of Time 54
3.7.2 Network Use 54
3.7.3 Quality and Correctness of the Code 54
3.7.4 Only One Software Platform 55
3.7.5 Node.js Event Loop and Garbage Collection 55
Chapter 4: Test Results 57
4.1 Idle Client Phase 57

4.1.1 CPU Load 58
4.1.2 Memory Footprint 59
4.1.3 Response Time 60
4.2 Test Phase - Scenario 1 60
4.2.1 CPU Load during Broadcast 61
4.2.2 Response Time during Broadcast 62
4.3 Test Phase - Scenario 2 63
4.3.1 CPU Load during Chat 63
4.3.2 Response Time during Chat 64
4.4 Memory Footprint after Tests 66
4.5 Result Summary 67
Chapter 5: Discussion 69
5.1 Idle Clients 69
5.2 Load Testing 69
5.2.1 The Long Polling Server Performs Expectedly Bad 71
5.2.2 The Server-Sent Events Server Performs Great 71
5.2.3 WebSocket is the Best Choice under Moderate Load 71
5.3 Stress Testing 71
5.3.1 Unexpected and Dramatic Increase in Response Time 73
5.3.2 Average vs. Median Response Time 73
5.3.3 Memory 73
5.4 Anomaly Discussion 75
5.4.1 Issue with the WebSocket Implementation 75
5.4.2 Node.js 75
5.4.3 Node.js' HTTP Is More Tested and Stable 76
5.4.4 Errors with the Test Implementation 76
5.5 Implementation 76
5.5.1 Test Scenario 1 77
5.5.2 Test Scenario 2 78
5.5.3 APIs 78
Chapter 6: Conclusion 79
6.1 Thesis Conclusion 79
6.1.1 Sub-Questions 79
6.1.2 Main Research Question 80
6.2 Further Work 81
6.2.1 Real-time Web at Scale 81
6.2.2 More Platforms 81
6.2.3 Node.js Scalability 81
6.2.4 Node.js Garbage Collection 81
6.2.5 HTTP 2.0 81

6.2.6 WebRTC 82
References 83
Appendix 87
Code 87
Software Versions 87
How to Run the Tests 88
Scenario 1 88
Scenario 2 88
Full Test Results 89
Idle Clients Phase 89
Test Phase - Scenario 1 92
Test Phase - Scenario 2 94
Memory Footprint after Test - Scenario 1 97
Memory Footprint after Test - Scenario 2 98

List of Figures

Figure 1: HTTP GET request to www.uio.no. 18
Figure 2: HTTP GET response from www.uio.no. 18
Figure 3: HTTP Polling example. 20
Figure 4: HTTP Long Polling example. 21
Figure 5: Issue with Long Polling requiring server logic. 22
Figure 6: The EventSource API. 23
Figure 7: The WebSocket API. 25
Figure 8: The WebSocket frame as defined in the RFC. 27
Figure 9: Relation between load and stress tests. 34
Figure 10: Expected response times. 35
Figure 11: The three components in the first test scenario. 38
Figure 12: The components in the second test scenario. 38
Figure 13: The three test phases and what is measured in each phase. 39
Figure 14: Example showing five clients sending their first two messages. 45
Figure 15: Sequence diagram for the first test scenario. 46
Figure 16: Sequence diagram for the second test scenario. 48
Figure 17: The WebSocket server in scenario 1. 51
Figure 18: The Server-Sent Events server in scenario 1. 51
Figure 19: The HTTP Long Polling server in scenario 1. 52
Figure 20: The WebSocket server in scenario 2. 53
Figure 21: The Server-Sent Events server in scenario 2. 53
Figure 22: The HTTP Long Polling server in scenario 2. 54
Figure 23: The three test phases and what is measured in each phase. 57
Figure 24: CPU load in the idle phase. 58
Figure 25: Memory footprint in the idle phase. 59
Figure 26: Response times in the idle phase. 60
Figure 27: CPU load in the first test scenario. 61
Figure 28: Response times in the first test scenario. 62
Figure 29: CPU load in the second test scenario. 63
Figure 30: Response times in the second test scenario. 64
Figure 31: Zoomed in response times in the second test scenario. 65
Figure 32: Memory footprint after the first test scenario. 66
Figure 33: Memory footprint after the second test scenario. 67
Figure 34: Response times in the first test scenario until the CPU was limited. 70
Figure 35: Response times in the second test scenario until the CPU was limited. 70
Figure 36: Response times in the first test scenario after the CPU was limited. 72
Figure 37: Response times in the second test scenario after the CPU was limited. 72
Figure 38: Memory footprint after the first test scenario. 74
Figure 39: Memory footprint after the second test scenario. 75
Figure 40: The three different servers in the first test scenario. 77
Figure 41: The three different servers in the second test scenario. 78

List of Examples

Example 1: Socket programming. 16
Example 2: A GET request example. 17
Example 3: Server-Sent Events connection. 23
Example 4: Server-Sent Events. 24
Example 5: How to open a WebSocket connection. 26
Example 6: WebSocket opening handshake. 27
Example 7: WebSocket connection with subprotocols. 28
Example 8: WebSocket extension. 28
Example 9: Stock price example JSON object. 29
Example 10: HTTP response with headers. 29
Example 11: HTTP request headers. 30
Example 12: Server-Sent Events message. 30
Example 13: Blocking code. 31
Example 14: Non-blocking asynchronous code. 32
Example 15: How to install a package to your Node.js project. 32
Example 16: How to allow 1024 user processes in OS X. 44
Example 17: Server-Sent Events Endpoint from the test code. 50

List of Tables

Table 1: Ranking of the three transports in both test scenarios. 68
Table 2: The software versions used in this thesis 87

Preface

This thesis is part of my Master's degree in Informatics: Programming and Networks from the University of Oslo. I worked as a part of the Programming and Software Engineering (PSE) research group at the Department of Informatics.

During my 2013 summer internship in the company BEKK, I was attending a presentation of cutting-edge web technologies that was held by Kim Joar Bekkelund. Later that summer, I contacted him over email, asking if he had any ideas for master thesis topics related to the Web. He proposed several interesting questions, but one stood out for me; comparing WebSocket to traditional HTTP methods for real-time functionality on the Web. Bekkelund also kindly agreed to be my primary supervisor. That was the start of this thesis.

All subjective views and opinions presented in this thesis represents only my personal opinions, neither that of the University of Oslo nor BEKK.

Acknowledgements

First, I would like to thank my primary supervisor from BEKK, Kim Joar Bekkelund for his great insight, support and cooperation. I would also like to thank my supervisor Yngve Lindsjørn, representing the University of Oslo, for his help in making this thesis properly academic.

A special thanks to my amazing girlfriend and best friend, Caroline. You always support me in the work that I do, no matter what. Your support, kindness, and warm presence give me strength.

I also want to thank my wonderful family who has supported me through all my years of education.

Johannes Akse, Joakim Lindquister and especially my father, Odd Tangen, provided extremely useful feedback after reading the thesis in its later stages. Thank you!

Finally, I would like to thank Kåre and Ragnhild Austdal for letting me stay in writing isolation at their home while writing the last parts of this thesis.

Chapter 1: Introduction

1.1 Motivation

The Web started out as a simple document sharing service to make the lives of researchers at CERN simpler. The transition into the Web of today has been made possible by the fact that it became capable of real-time updates. The real-time web enables a website to update itself dynamically when new content is available.

The following two examples illustrate the meaning of the term *real-time* in this thesis:

1. A stock price application where several clients connect to a server. This stock price server subscribes to a broker backend for stock price updates. When the stock price server receives price updates from the backend, it broadcasts these to all clients.
2. A chat room application where several clients connect to a centralized server. The server listens for client messages and as soon as a client sends a message to the server, the server immediately broadcasts the message to all other connected clients.

Both examples show that data is handled and distributed immediately after it is received, which is what is meant by the term real-time in this thesis.

The first example requires unidirectional messaging, with stock prices only going server-to-client. The second example is bidirectional, requiring both server-to-client and client-to-server messaging. This thesis will focus on both types of real-time applications.

Even though real-time capabilities were not considered when the Web was designed, there are several ways to use traditional HTTP in order to achieve near real-time updates, such as the techniques *Long Polling* and *Streaming*. The problem is that these have difficulties, which Chapter 2 explains in detail. Recently, with the new protocol *WebSocket* and the new HTTP-based technology *Server-Sent Events*, a true real-time web is possible. This thesis will therefore focus on WebSocket, Server-Sent Events, and HTTP Long Polling.

Server-Sent Events and Long Polling are both unidirectional (server-to-client) while WebSocket is bidirectional (server-to-client and client-to-server). These will be presented in turn in Chapter 2.

For simplicity's sake, this thesis will refer to the real-time delivering technologies as *transports*.

It is believed that WebSocket performs better than HTTP [1] [2]. However, it would be interesting to see if that is true when running full performance tests (more on performance testing in Section 2.9) while comparing it to Server-Sent Events and Long Polling. Kristian Johannessen's master's thesis [3] indicates that WebSocket is the most performant, although he had a focus on frameworks and not the transporting technologies themselves.

In addition to the performance aspect, the transports are different conceptually. Therefore, it would be interesting to see how well they fit into a real-time application from a programmer's perspective.

1.2 Research Questions

The goal of this thesis is thus to compare WebSocket to the two HTTP real-time transporting technologies Server-Sent Events and Long Polling. They will be compared based on two grounds; performance and programmer friendliness in a real-time setting.

With this basis, the primary research question of the thesis is:

- For what types of real-time web applications does WebSocket provide a benefit over Long Polling and Server-Sent Events?

To substantiate and validate this, I have formed three additional sub-questions:

1. How does WebSocket perform compared to Long Polling and Server-Sent Events in a unidirectional, server-to-client messaging setting with high client load levels?
2. How does WebSocket perform compared to Long Polling and Server-Sent Events in a bidirectional messaging setting with high client load levels?
3. Does WebSocket provide any advantages over Long Polling and Server-Sent Events in a real-time setting, from a programmer's perspective?

1.3 Thesis Outline

Chapter 2 presents all background material needed to understand the methodology and research in this thesis. That means all three transports, the chosen software platform and a description of performance testing. The chapter starts with a section on relevant academic works.

Based on the background material, I have developed two distinct real-time scenarios to compare the three different transports. Chapter 3 presents these two scenarios and discusses parameters and choices made. It also includes a section on how the test results are collected.

Chapter 4 presents the test results in a descriptive manner, while Chapter 5 discusses and analyzes the main observations.

Chapter 6 contains the thesis conclusion that directly answers the research questions. The chapter concludes with a section on further work.

Chapter 2: Background

This chapter will present all technologies that the real-time web involves. That includes the protocols TCP, HTTP, and WebSocket, as well as HTTP-based techniques like Long Polling and Streaming. This chapter also includes material on the development platform Node.js and some basics of performance testing. First, I present a section on related academic work.

2.1 Related Academic Work

In the 2007 paper Bozdag et al. [4] compare different real-time techniques for the Web. The two concepts in question are server-pull and server-push. Server-push means that server updates are pushed from the server to the client, while server-pull means that the client actively asks (pulls) for updates. The server-pull approach requires extra server logic as well as more network traffic compared to server-push. Because of this, the authors conclude with praise for the push approach if high data coherence and network performance is desired. But they do point out some problems concerning scalability. The push method they tested and thus recommended was HTTP Streaming (presented in detail in Section 2.5.2). Server-Sent Events and WebSocket are other push techniques, but are not part of the paper, as they did not exist at the time when it was written.

The bachelor's thesis by Jõhvik [5] seems to be inspired and motivated by the work of Bozdag et. al. Jõhvik points out some drawbacks of HTTP Streaming, such as the potential for memory leaks and the lack of auto-reconnect. After testing, he determines that HTTP Streaming does not perform well enough to justify a choice over HTTP Long Polling, conflicting with the views of the previous article. WebSocket and Server-Sent Events did exist at the time of the article's writing, but Jõhvik decided not to include them because of web browser incompatibility.

In the first part of the thesis of Kristian Johannessen [3], he compares different real-time frameworks for the Web. Some of the frameworks such as SignalR [6] and Lightstreamer [7] support several real-time transporting techniques. Some even provide fallback solutions to support the largest possible set of clients. Based on performance and developer friendliness (maturity, web browser support, WebSocket support, and presentation), he recommends SignalR as the best real-time framework, closely followed by Socket.IO [8].

The second part of his thesis compares WebSocket to traditional HTTP techniques for real-time behavior. He concludes by saying "WebSocket is better than HTTP in every aspect of real time applications," although he is surprised by how well Server-Sent Events performs in server-to-client communication.

Johannessen's and my thesis differ in that he had a focus on frameworks under moderate load levels, while I focus on transports and high load levels.

The rest of this chapter contains a description of the protocols HTTP and WebSocket, as well as HTTP techniques such as HTTP Long Polling for real-time behavior. Node.js, the

chosen software platform, is also presented. And finally, there is a section on performance testing. I will start with the Transmission Control Protocol (TCP).

2.2 TCP

To better understand how HTTP and WebSocket work, it is essential to have a basic understanding of TCP, even though the protocol in itself is not directly used in this thesis. TCP is one of the most important protocols on the internet, as it forms the foundation of the Internet Protocol Stack [9].

The most important aspects of TCP are:

- TCP is a *transport layer protocol*. The transport layer is the layer below the application layer where HTTP and WebSocket lie.
- TCP is *connection-oriented*. This means that a connection has to be established before any exchange of data. Once the connection is established, users can push data at any time.
- TCP is *reliable*. This means that every packet will eventually arrive at the receivers end. This is ensured by having the receiver acknowledge of each received packet.
- TCP is *bidirectional* and *full-duplex*, meaning that both parties can communicate at the same time, whenever they want.

A TCP connection's endpoints are called sockets. A socket is a data structure abstraction that can be both written and read from and treated like a file. This type of programming is called socket programming. With being full-duplex, TCP makes it easy to build real-time applications with immediate data push behavior. Example 1 shows pseudocode (and a simplification) that demonstrates how easy it is to connect to a remote server and immediately send data over a socket.

```
var socket = SocketLibrary.connect(remote_host_address);
socket.send("data");
```

Example 1: Socket programming.

2.3 The Web and HTTP

2.3.1 The World Wide Web

The Web was originally designed to fetch static, non-styled, text-only documents. Over time style sheets and script files were added and today the Web mainly consists of these three components:

- HTML - An XML-like markup language that describes a website's content.
- CSS - A language that describes styling attributes of HTML components.
- JavaScript - The web's programming language.

The web still uses the basic principle of document fetching. But the "documents" retrieved by a web browser can be highly complex and interactive applications, with Google Maps as an example. Google Maps is completely different from simple websites such as blogs or

newspapers but is powered by the same technologies. Today it is even likely that HTML, CSS and JavaScript power applications running locally on your smartphone.

As we will see later in this chapter, there is one area where the Web has lagged far behind platform-native applications; the networking protocols. HTTP works great for simple document fetching but it is not designed for the advanced use cases of today's web applications. As this thesis will reveal, it is hard and suboptimal to develop real-time applications using HTTP.

HTML5 [10] intends to improve web transports with Server-Sent Events and WebSocket. Server-Sent Event extends HTTP and gives the ability to push data natively from the server. WebSocket is a new protocol, bringing TCP-like socket programming to the Web.

2.3.2 HTTP

HTTP is short for HyperText Transfer Protocol and is the protocol used to deliver web pages to a user's web browser. The protocol is request-response oriented which implies that all server-sent messages must be a response to a certain request. HTTP has many types of requests, called *methods*.

2.3.3 HTTP Methods

There are several HTTP request methods, but the most important ones are GET, HEAD, POST, PUT, and DELETE. Requests methods are sent as plain ASCII text, and the server parses them and responds with the requested information. Each response is marked with a status code to indicate whether the request was successful or not.

GET was the first HTTP method [11] and is the one you send to a web server to request a certain file or document. Example 2 shows a simple GET request to http://www.uio.no.

```
GET /index.html HTTP/1.1
Host: www.uio.no
```

Example 2: A GET request example.

The request consists of a GET followed by the document's address. The server parses the GET request and sends the requested document back as a HTTP response.

POST is used in conjunction with web forms and when a user wants to submit data to the server. DELETE is used to inform the server to delete a certain resource. HEAD is used as a GET where you do not want the actual response data, but only the response *header fields*.

2.3.4 HTTP Header Fields and State

HTTP is a stateless protocol. That means the server does not store any information regarding the clients, which is a great thing in terms of server resources. But it can also be a problem, as we shall see later on.

Header fields are an important part of HTTP. They function as metadata that are added into the HTTP requests and responses. The «Host» line in Example 2 is the Host header field.

Headers are there for the server and client to give the other part some necessary information. As an example, it is useful for the server to know what kind of language the client understands.

Figure 1, captured by HTTP Scoop [12], shows the entire GET request a web browser sends when going to www.uio.no.

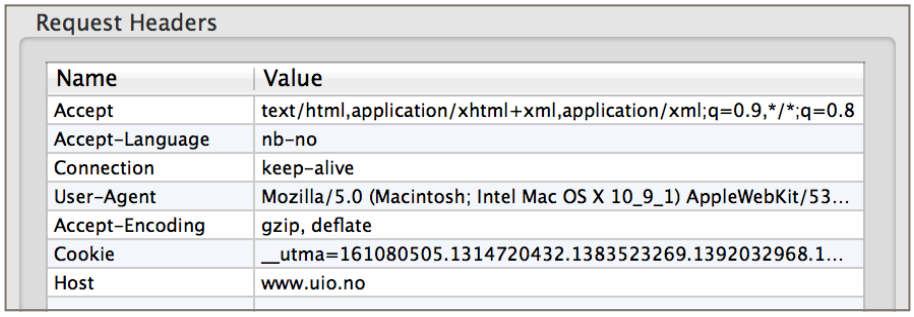

Figure 1: HTTP GET request to www.uio.no.

The web browser adds several header fields to each request. For example, the User-Agent field tells the server what kind of computer, operating system and web browser the client is running, while the Accept header tells the server what file types the client can read. The Connection: keep-alive field means that the user wants the server to keep the underlying TCP socket open, as it is likely to send more requests soon. HTTP Headers are an important part of the protocol and adds a slight sense of state to the otherwise stateless protocol.

The server responds to a request with an HTTP response. The response contains header fields followed by the HTML code for the website. As the web browser parses and renders the HTML file from top to bottom, it may find link, script and image tags inside the markup. This means that the website consists of more elements and that the client must request those as well. As an example, for http://www.uio.no there was a total of 56 files (JavaScript, CSS, and image files) to be fetched, resulting in 56 GET requests and 56 server responses. Figure 2 shows an example of a server response with its headers.

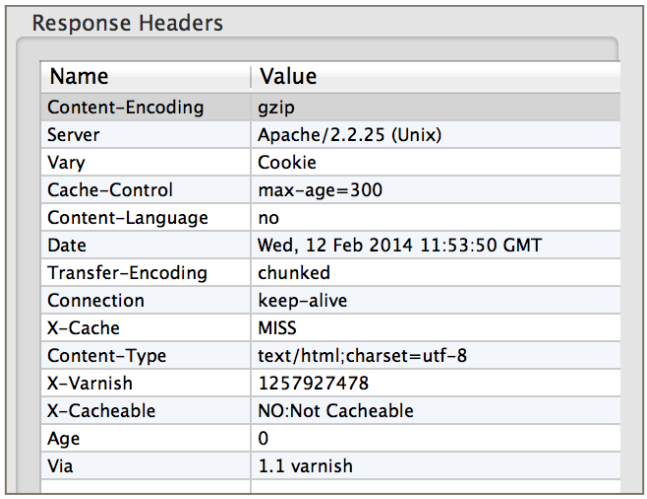

Figure 2: HTTP GET response from www.uio.no.

2.4 Modern Web

HTTP was designed to serve static hyperlinked documents, but today you rarely visit a website that is static and pure HTML. Most of the websites you visit are highly interactive with JavaScript code running in the background. This development started in the late 1990s with Microsoft Outlook and was truly utilized by Google Gmail and Google Maps in 2004. These types of websites started to behave more like platform-native applications and were a definite step away from the hyperlinked documents that the Web originally was. With more sophisticated websites and much JavaScript code, you needed a faster web browser. It became essential for the main web browser vendors Microsoft, Google, Mozilla and Apple to build fast JavaScript engines.

Essential to this development was how HTTP was used in order to achieve dynamic updates; using Asynchronous JavaScript and XML (Ajax).

2.4.1 Ajax

In 2004, Jesse James Garret introduced Ajax [13]. He stated that Ajax is "several technologies, each flourishing in its own right, coming together in powerful new ways". A central aspect of Ajax is the *XMLHttpRequest* JavaScript API [14]. It is used to send and retrieve data from a server asynchronously, using HTTP. Previously, a web browser typically requested the entire website for each GET it sent. With Ajax, this server interaction occurs in the background and the client-side JavaScript updates the DOM (Document Object Model: the HTML view) with new data.

Even though Ajax has XML in its name, the data type is not limited to just XML. JSON [15] is a format for representing hierarchical key-value data with less overhead compared to XML, and is widely used in conjunction with Ajax.

An example of an Ajax-powered web application is Google Maps. When you pan around the map, the JavaScript running in your web browser initiates Ajax GET requests to the server, requesting data of the area you are now looking at. When new images and map data arrives, JavaScript running in your web browser updates the DOM.

Ajax is essential to web applications, and it brought interactivity to an otherwise static web. Together with some techniques (as Long Polling and Streaming, described below), Ajax can make the Web real-time.

2.5 Real-Time HTTP

For many applications, pushing data between a server and a client is essential. An example is a web application displaying stock prices. Stock prices can change very often, many times per minute. As soon as the server receives a stock price update from the broker, it would like to push the update immediately to the connected clients. Achieving this kind of push behavior is quite trivial for platform-native applications because you can set up a full-duplex TCP socket and push updates as they arrive. Even though HTTP utilizes TCP on the transport layer, HTTP itself is just half-duplex. That means only one side can send data at a time, and there

is no way to push natively from the server. All server-sent messages must be a response to a client-sent request.

To get the real-time behavior we see today with Twitter's feed and Facebook's chat using HTTP, developers had to utilize the techniques presented below.

2.5.1 HTTP Polling

The first solution (as showed in Figure 3) is having the client-side JavaScript periodically poll the server for updates. If these requests are sent frequently enough, it could be *perceived* as real-time. This approach is called *HTTP Polling* and is quite simple conceptually and straightforward to implement. HTTP Polling works ideally if you know exactly when the server updates its data, and you can ask for new values directly after that. That is, however, rarely the case. Take a chat application as an example; you do not know when the one you are chatting with sends a message. It can vary from a couple of seconds to several minutes if the message is long. Trying to find the perfect request rate is difficult and varies greatly from application to application. The worst case scenario is that you end up sending a lot of requests that return empty responses. This side effect is certainly a bad thing, as it congests the network with unnecessary traffic.

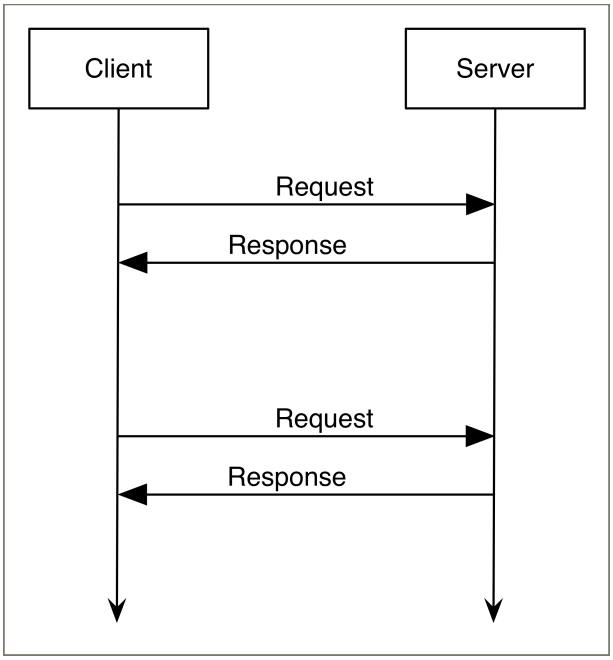

Figure 3: HTTP Polling example.

2.5.2 Comet

Comet is an umbrella term for a set of programming models that achieve server-push behavior using existing HTTP technologies. The term Comet was first introduced by Alex Russell in a blog post [16] he wrote in 2006. The two most used Comet techniques are HTTP Long Polling and HTTP Streaming.

HTTP Long Polling (Figure 4) is essentially the same as regular HTTP Polling except that the server delays the response until either new data is ready, or when a timer runs out. Immediately after the response is received, the client sends a new request and waits for new

updates. As a default, the timer is 45 seconds [4]. Long Polling gives the user the impression of having data pushed from the server, even though it, in theory, is not.

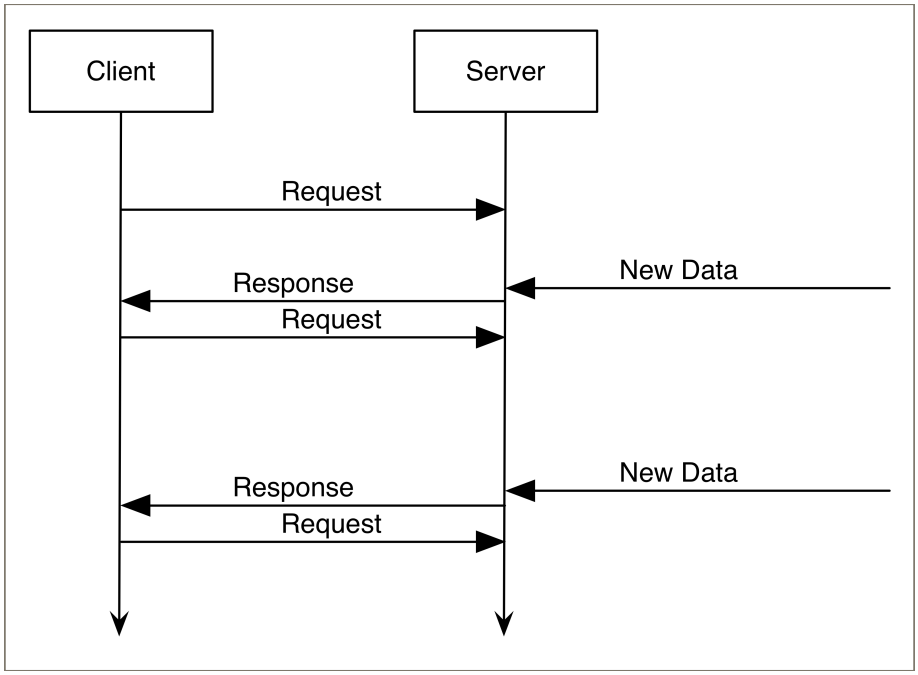

Figure 4: HTTP Long Polling example.

HTTP Streaming, also known as "the forever-frame," is another technique that emulates server-push. Chunked Encoding is a part of the HTTP/1.1 specification that lets the server push chunked data to the client before the response size is known. A forever-frame is an HTML iframe that keeps receiving scripts as these chunks. The scripts are immediately executed on the client-side, and the server can in practice keep this connection open as long as it wants.

2.5.3 Why Comet and HTTP is Unsatisfactory

If both Long Polling and Streaming give web applications real-time push of data, why is there a need for Server-Sent Events and WebSocket? HTTP Streaming suffers from several problems, such as a potential for memory leaks, proxy issues and no support for auto-reconnect [5]. With scripts being immediately executed, security is also a concern. The scripts can contain harmful code.

For HTTP Long Polling, consider our stock application from earlier, but now with clients using Long Polling. Between the Long Polling timer runs out and a new request is sent from the client, a new price has arrived from the stock broker (example in Figure 5). Now the server must remember that this specific client has outdated information and push data as soon as the next polling request arrives. This adds complexity to an otherwise simple task. It even breaks the idea that the server should stay stateless.

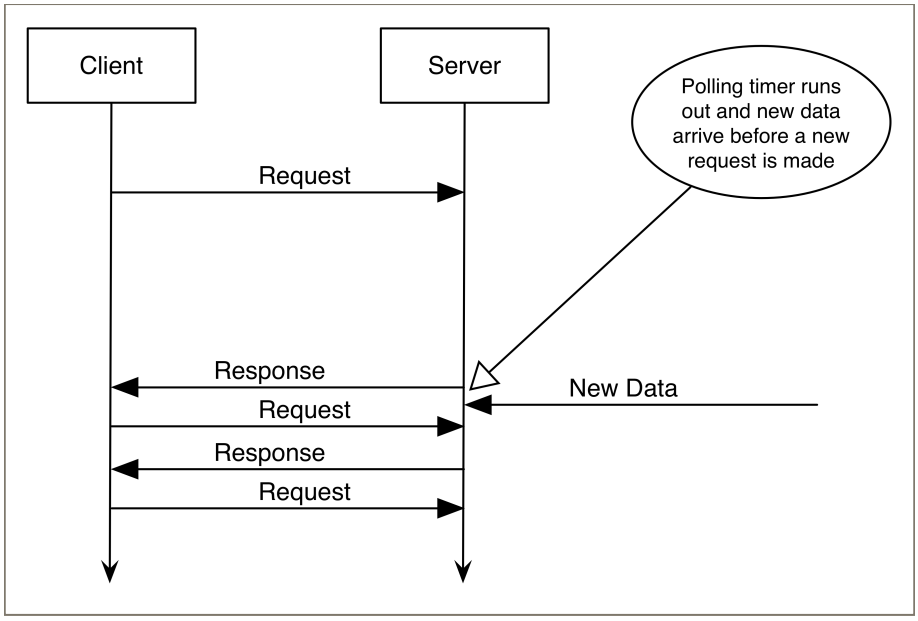

Figure 5: Issue with Long Polling requiring server logic.

Another issue is related to HTTP headers. In The GET example to http://www.uio.no from Subsection 2.3.4, the amount of header data for each request was between 500 and 800 bytes. Additionally, all 56 response headers were between 300 and 500 bytes. Many real-time applications only send small messages, maybe just a couple of bytes. This vast amount of unnecessary header data is repeated for each request and can congest the network.

And importantly, all Comet techniques are for server-to-client messages only. If a real-time web application requires client-to-server messages as well (i.e., a chat application), you need to accommodate for those using other techniques. Because HTTP is stateless, server-to-client and client-to-server messages must take place independently of each other. That adds to the server complexity.

Long Polling and HTTP Streaming accomplish push behavior, but they have several disadvantages. HTTP was designed for an outdated web and it does not utilize the full potential TCP gives. All the problems presented here, could have been fixed by a connection-oriented protocol like TCP.

2.5.4 HTML5

HTML5 is the fifth revision of the HTML markup language and the first major update since HTML4 was standardized in 1997. Even though HTML5 adoption started many years ago, the W3C recommendation was just recently finalized [17]. HTML5 is, despite its name, much more than just an updated HTML version. It is a collection of many technologies that intends to clean up the syntax and unify web technologies. It also introduces several new APIs that make the Web a platform for full-fledged applications. Because of the lack of native real-time capabilities in HTTP, HTML5 introduces Server-Sent Events and WebSocket.

2.6 Server-Sent Events

Server-Sent Events is a new HTTP technology that solves the issues with server-push from Subsection 2.5.3. It does this by providing the concept of a long-lived connection and a nice programmer interface. As its name implies, Server-Sent Events is unidirectional, supporting server-to-client messages only. To understand Server-Sent Events, we must understand the EventSource API [18] and the Event Stream Protocol.

2.6.1 EventSource API

```
[Constructor(in DOMString url)]
interface EventSource {
  readonly attribute DOMString URL;

  // ready state
  const unsigned short CONNECTING = 0;
  const unsigned short OPEN = 1;
  const unsigned short CLOSED = 2;
  readonly attribute unsigned short readyState;

  // networking
           attribute Function onopen;
           attribute Function onmessage;
           attribute Function onerror;
  void close();
};
EventSource implements EventTarget;
```

Figure 6: The EventSource API.

As seen in Figure 6, the EventSource API is small and simple, yet powerful. It provides different ready states (CONNECTING, OPEN and CLOSED) to make the connection stateful. It also offers three events which a client can listen for (open, message and error). A simple close method is added for connection teardown. Example 3 shows how simple it is to open up a connection and listen for server messages.

```
var source = new EventSource('http://example.com/sse');
source.onmessage = function(m) {
     // Code to be executed once a message has arrived
}
```

Example 3: Server-Sent Events connection.

2.6.2 Event Stream Protocol

Under the hood, Server-Sent Events is implemented as HTTP Streaming over a long-lived HTTP connection. In addition to the simple API, other advantages over regular HTTP Streaming include automatic reconnects and message parsing [19].

Syntactically, what make Server-Sent Events different from HTTP Streaming are the Accept and Content-Type header fields. The new value is "text/event-stream". Example 4 illustrates how this exchange is done and how data is sent from the server.

HTTP Request:

```
GET /stream HTTP/1.1
Host: example.com
Accept: text/event-stream
```

HTTP Response:

```
HTTP/1.1 200 OK
Connection: keep-alive
Content-Type: text/event-stream
Transfer-Encoding: chunked

retry:      15000

data: First message is a simple string.

data: {"message": "JSON payload"}

id: 42
event: bar
data: Multi-line message of
data: type "bar" and id "42"
```

Example 4: Server-Sent Events.

In Example 4 you can see how the server sets the client reconnect interval to 15 seconds. The example is part of an example found in the book High Performance Browser Networking [19] and shows that the data format can be pure text or JSON. Other features include the ability to set an id and a custom event associated with a message.

2.6.3 Server-Sent Events Problems

With a simple API and developer friendly features like automatic reconnects, Server-Sent Events should be the obvious choice for developers wanting server-push on the Web. Sadly that is not the case. The way I see it, there are two reasons for it. First and in some cases, not very important; you can only send string data. If you need to send binary it has to be converted using base64 encoding. This adds some overhead. Second, the adoption is not perfect. All modern web browsers except Internet Explorer support Server-Sent Events. Because Internet Explorer accounts for a large portion of the web browser market, choosing Server-Sent Events alienates many users.

While many applications benefit from Server-Sent Events, some require client-to-server messaging as well. Server-Sent Events alone has no answer to that need. But as we shall see, WebSocket does.

2.7 WebSocket

Unlike Server-Sent Events, WebSocket is an entirely new protocol for the Web. WebSocket is an application layer protocol with full-duplex communication support. It utilizes a single TCP socket but simplifies some of the underlying protocol's rough edges. WebSocket promises to be all about performance, simplicity, standards and HTML5 [20]. It is designed to work seamlessly together with HTTP. In order to understand what is unique to WebSocket and why it is important, we must dig into two parts; the protocol itself (RFC 6455 [21]) and the API.

2.7.1 The WebSocket API

One of the great powers of WebSocket is its simple, yet powerful JavaScript API. Figure 7 shows the entire interface.

```
[Constructor(DOMString url, optional (DOMString or DOMString[]) protocols)]
interface WebSocket : EventTarget {
        readonly attribute DOMString url;

        // ready state
        const unsigned short CONNECTING = 0;
        const unsigned short OPEN = 1;
        const unsigned short CLOSING = 2;
        const unsigned short CLOSED = 3;
        readonly attribute unsigned short readyState;
        readonly attribute unsigned long bufferedAmount;

        // networking
                        attribute EventHandler onopen;
                        attribute EventHandler onerror;
                        attribute EventHandler onclose;
        readonly attribute DOMString extensions;
        readonly attribute DOMString protocol;
        void close([Clamp] optional unsigned short code, optional DOMString reason);

        // messaging
                        attribute EventHandler onmessage;
                        attribute DOMString binaryType;
        void send(DOMString data);
        void send(Blob data);
        void send(ArrayBuffer data);
        void send(ArrayBufferView data);
};
```

Figure 7: The WebSocket API.

Similarly to the Server-Sent Events API, we see several ready states and networking events to listen for. We also see the *extensions* and *protocol* attributes. What they are will be discussed further on. WebSocket is full-duplex, meaning it can send messages as well as receiving, at any time. Sending is done with the send methods. These methods accept either String or binary data. Because WebSocket is a new protocol, strings are expected to be coded in UTF-8, removing all encoding problems. You call the close method when you want to terminate the connection.

Example 5 shows how to open a WebSocket connection.

```
var ws = new WebSocket('ws://example.com');
ws.onopen = function(e) {
      // Code to be executed once the connection is established
}
```

Example 5: How to open a WebSocket connection.

2.7.2 The WebSocket Protocol

The WebSocket protocol was designed to work seamlessly together with HTTP. In fact, WebSocket uses an HTTP request's Upgrade header field to tell the server that it wants to upgrade from HTTP to WebSocket. This is all done over the same ports as HTTP to provide a simpler rollout of the protocol. This upgrade is part of what is called the WebSocket opening handshake.

WebSocket opening and closing handshake

To open a WebSocket connection, a client sends an HTTP request to the server with the header field *Upgrade: websocket*. The server responds to this request with a 101 status code and the same header field in return. The 101 status code indicates that the server is switching protocols. Once the client receives this response, the open event is triggered, and the connection is established. This short exchange of HTTP packets is the WebSocket opening handshake.

Similarly to the opening handshake, WebSocket also has a closing handshake. This handshake is there to differentiate between intentionally and unintentionally teardowns of the connection. As the API described, the user can send a status code and a UTF-8 text string to tell the server why the connection was closed.

Figure 6 shows the WebSocket opening handshake. The exchange of keys is happening to ensure both parties speak the same WebSocket version.

HTTP Request:

```
GET /chat HTTP/1.1
Host: server.example.com
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Key: x3JJHMbDL1EzLkh9GBhXDw==
Sec-WebSocket-Version: 13
Origin: http://example.com
```

HTTP Response:

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: HSmrc0sMlYUkAGmm5OPpG2HaGWk=
```

Example 6: WebSocket opening handshake.

Message Format

To keep the API simple, WebSocket abstracts away some of the roughness of TCP. When you want to send a message over a TCP socket, the message could be divided into several chunks, depending on its size. As a developer, you have to deal with the fact that they are delivered as chunks and not as whole messages. WebSocket takes care of this for you, and the message event is only triggered once an entire message is delivered. Even though the protocol abstracts away the framing for the developer, messages are indeed sent as chunks (frames). Figure 8 shows a WebSocket frame.

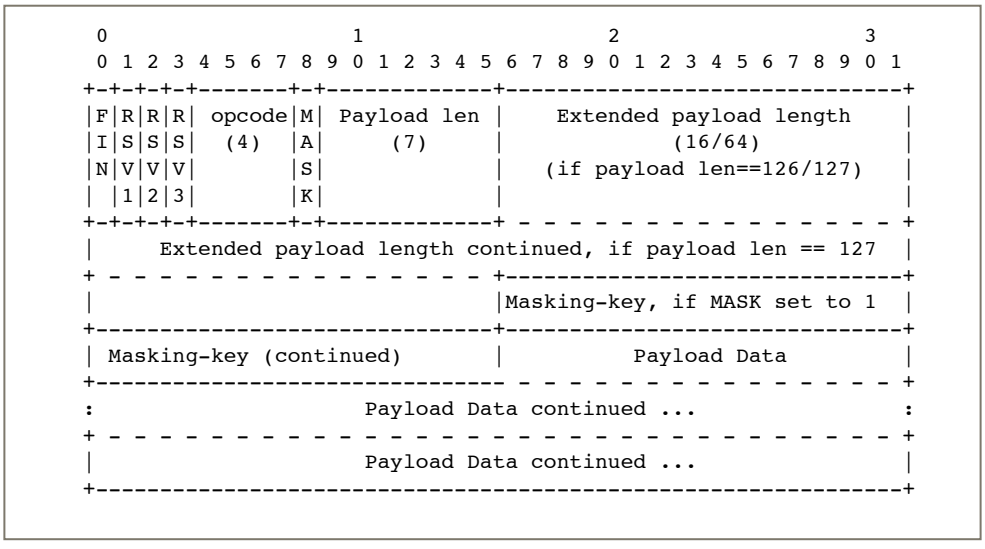

Figure 8: The WebSocket frame as defined in the RFC.

For this thesis, most of the fields in a WebSocket frame are not that important. But I want to show it nevertheless because it illustrates a big difference to HTTP. Look at how the payload length field is found in three places. This means support for a variable number of bits

denoting the payload length. If the frame is between 0 and 126 bytes, only 7 payload length bits are need. For payloads between 126 and 216 an extra two bytes (7 + 16 bits) are added and for larger frames an additional 8 bytes (7 + 64 bits) are added. For very small messages, only *3 bytes* of header data is necessary. Compare this to HTTP where each message needs many header fields, each consisting of tens of bytes.

Subprotocols and Extensions

With its simple, yet powerful API, WebSocket is built to enable higher level protocols and frameworks to be built on top of it. These higher level protocols are called *subprotocols*. When establishing a WebSocket connection you can pass in an array of subprotocol names.

```
var ws = new WebSocket('ws://example.com', ['proto1', 'proto2']);
```

Example 7: WebSocket connection with subprotocols.

In Example 7, the client tells the server at example.com that it speaks both 'proto1' and 'proto2.' If the server knows these, the server can choose which one to use, but only one at a time. There is support for several official protocols [22], such as Microsoft SOAP and unofficial open protocols such as XMPP. And it is possible for anyone to create WebSocket subprotocols.

Extensions represent another way to append WebSocket with additional features. Unlike subprotocols, you can extend your WebSocket connection with several extensions. An extension is a supplement to the already existing protocol, and both web browser and server must support it. You add extensions with the Sec-WebSocket-Extension header. Example 8 is an extension that compresses frames at source and decompresses at destination.

```
Sec-WebSocket-Extensions: deflate-frame
```

Example 8: WebSocket extension.

2.7.3 WebSocket and HTTP

WebSockets are great, but will not replace HTTP. Instead, the two protocols will work together to bring tomorrow's real-time web applications to market. There are features of HTTP which WebSockets do not provide. It does not make sense to download all website assets over WebSocket, as HTTP already has caching abilities. Cookies are another part of HTTP not available to WebSockets. And, even though HTTP being stateless can be a bad thing, it can also be good. Statelessness means there is no need for additional server resources beyond the ones already allocated for the HTTP request.

WebSocket is an easy-to-use, modern and powerful TCP-like protocol. In my opinion it even improves upon TCP, with its simple subprotocol scheme and frames being abstracted away. The web has finally caught up with platform-native applications in terms of real-time networking capabilities.

2.7.4 WebSocket vs. Server-Sent Events and HTTP

One of the issues with HTTP was the large amount of header data. With my HTTP GET example to http://www.uio.no, every request and response had several hundred bytes of metadata.

Since WebSockets are stateful, message sizes can be tiny in comparison. To illustrate this difference, I have created an example based on the stock price application from Section 1.1. How do HTTP Long Polling, Server-Sent Events, and WebSocket solve server-push? A stock price update is represented by a 58 byte long JSON object with three attributes; the message type, the price update and the time (Unix timestamp) the price was updated. Example 9 shows the object.

```
{
      "type": "priceUpdate",
      "price": "24.45",
      "time": "1429528134"
}
```

Example 9: Stock price example JSON object.

HTTP Long Polling

In addition to the 58 byte long JSON object we want to send, an extra 221 bytes are consumed by HTTP headers, totaling at 279 bytes per stock price update (see Example 10). The headers consume almost four times as much data as the short message we want to send.

Response Headers

```
HTTP/1.1 200 OK
X-Powered-By: PHP/5.4.0
Server: Apache/2.4.1 (Unix)
Date: Mon, 20 Apr 2015 13:33:28 GMT
Last-Modified: Mon, 20 Apr 2015 12:33:28 GMT
Content-Type: application/json
Content-Length: 63
Connection: keep-alive
```

Response Body

```
{"type":"priceUpdate","price":"24.45","time":"1429528134"}
```

Example 10: HTTP response with headers.

And importantly, HTTP is half-duplex, meaning every HTTP response follows an HTTP request. Assume that each HTTP request looks like the 263-byte long Example 11.

Request Headers

```
GET /poll HTTP/1.1
Host: example.com/stock
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:12.0) Gecko/
            20100101 Firefox/21.0
Accept: application/json
Accept-Language: en-us
Accept-Encoding: gzip, deflate
Accept-Charset: utf-8
Connection: keep-alive
Keep-Alive: 300
```

Example 11: HTTP request headers.

Now every stock price update requires 263 (request headers) + 279 (response headers and response body) = *542 bytes*.

Server-Sent Events

Server-Sent Events is connection-oriented, so there is no need for an HTTP-like request once the connection is up. There is not much wasted space on headers either. The server-pushed message can look like Example 12.

```
id: 1
data: {“type”:“priceUpdate”,“price”:“24.45”,“time”:“1429528134”}
```

Example 12: Server-Sent Events message.

Including a blank line at the end, each stock price message totals at *74 bytes* with Server-Sent Events. Only 16 bytes of header data is a vast improvement over HTTP.

WebSocket

Because the stock price update is just 58 bytes long and well below 126 bytes, WebSocket requires only 3 bytes of header data (see Subsection 2.7.2 under Message Format). As a result, each stock price update sent from a WebSocket server requires only *61 bytes*. Part of what makes the headers so small is the fact that they are binary encoded compared to the ASCII text based HTTP and Server-Sent Events ones.

2.8 Web Platforms & Node.js

2.8.1 Web Development

When developing for the Web, you need to develop on two distinct ends; the frontend and the backend. Unlike platform-native applications, the web frontend is limited when it comes to development choices. Your code has to be JavaScript, HTML and CSS.

On the backend however, you can freely choose web frameworks and programming languages. Traditionally Java and .NET frameworks such as Spring [23] and ASP.NET [24] have been very popular. Even though the clear separation of front- and backend works fine, a new platform called *Node.js* shows that there was a need for a more unified web development process.

2.8.2 Node.js

Ajax made web applications more complex, and developers spent more time writing JavaScript. The context switch from frontend JavaScript to another language on the backend could be cumbersome. When the creator of Node.js Ryan Dahl introduced server-side JavaScript [25] in 2009, many developers found the idea promising.

Node.js is a JavaScript runtime environment built on top of Google Chrome's V8 JavaScript engine. V8 is mostly written in C++ [26], meaning Node.js runs directly on the hardware. That makes it fast.

In addition to JavaScript on the server, Node.js brings some new features to server-side web development:

- Non-blocking code.
- Single threaded development environment.
- The lightweight package manager NPM.

With traditional threaded web servers, a new thread is spawned for each newly connected client. The server context switches between all threads and runs their code. However, most of the time, web servers are doing I/O, typically querying a database or reading a file. I/O operations block the running thread, and the server has to wait for the I/O operation to finish. This takes up precious CPU cycles, and the server compensates by doing context switches between threads. The problem is that context switches are expensive and threads take up memory.

Node.js breaks the threaded programming paradigm with something called an Event Loop. The event loop is an ever-going loop that constantly looks for triggered events. Examples of events can be a newly connected client or an answer to a database query. The event loop lets you program in a single-threaded environment that takes full advantage of the CPU. Because of the event loop, Node.js has proven to scale quite well [27].

To show how the two different programming styles compare, consider Example 13 and 14.

```
var result = database.query("some query"); // Code blocks here
// Result is fetched
something else;
```

Example 13: Blocking code.

```
database.query(“some query”, function(result) {
      // Result is fetched
});
something else;
```

Example 14: Non-blocking asynchronous code.

In Example 13 you can see that the first line blocks the following lines until the database query result is stored in the variable *result*. This is how it is to program in a threaded and synchronous environment. Most programming languages such as Java follow this model.

Example 14 shows how you typically write Node.js code. The difference here is that we send in a *callback* function to the query function itself. The callback function is called whenever the database has responded and is triggered by the event loop. The code following the database query can execute immediately.

Programming in an asynchronous manner is fundamentally different to the synchronous style most backend programmers are used to. Frontend developers, on the other hand, have been programming like this for some time. Ryan Dahl said during his Node.js introduction that JavaScript is the perfect language for a non-blocking environment. The web browser already has an event loop constantly listening for events such as button clicks. Node.js utilizes this and unifies web development around one programming style and one language.

NPM [28] (Node Package Manager) is another noteworthy feature of Node.js. NPM makes it easy to install packages for you to use in your projects. It works like shown in Example 15.

```
$ npm install <package-name>
```

Example 15: How to install a package to your Node.js project.

A final part of Node.js (V8 specifically) that needs to be understood, is how memory is managed. V8 employs a garbage collector. Memory allocation and deallocation are then handled by the runtime. There are two sides to garbage collection. First, it is wonderful for a programmer to not worry about memory management. Memory related issues, like leaks, are then less likely to happen.

Second, a garbage collector introduces a performance penalty. V8 uses the stop-the-world collection scheme [29]. That means V8 stops all program execution once the garbage collector runs. As a result, the user response time will be affected for the time the collector runs. That makes the performance to a certain degree less predictable, as the garbage collector can decide to run at unpredictable times.

There are tools to memory profile your Node.js application, but because Node.js is a new platform, the tools are not as mature as they are in other environments, such as the Java Virtual Machine.

2.9 Performance Testing

2.9.1 Introduction to Performance Testing

To determine what technology is the most efficient under a set of defined criteria, we can carry out performance tests. As stated in the book Performance Testing Guidance for Web Applications, "Performance testing is a type of testing intended to determine the responsiveness, throughput, reliability, and/or scalability of a system under a given workload" [30]. For a product launch on the internet, it is vital to know whether your systems can withstand the expected workload, especially on launch day. Testing is therefore, crucial and should be an integral part of software development. Performance testing can also help you identify bottlenecks in your system and assist you in building the most efficient solution possible. The book introduces two subcategories of performance testing:

Load Testing

Load testing is a type of performance test focused on determining performance qualities for a system that is under normal workloads.

Stress Testing

Stress testing is a type of performance test focused on determining performance qualities for a system under unnatural high workloads. That can include limited memory or processor resources.

In addition to the two types above there are other types of performance tests as well:

- Soak testing: This type of test is usually done to determine memory leaks. To get an accurate system leakage picture, a soak test usually runs for a long time.
- Spike testing: Spike tests are conducted to see how a system reacts to sudden spikes of workload.

I will only focus on load and stress tests in this thesis. For simplicity's sake, I have defined Load and Stress testing to mean the following:

Load testing: As long as the server CPU usage is below the maximum, the test is a load test.

Stress testing: When the server CPU load is at a maximum, the test is a stress test.

Figure 9 shows how a load test "becomes" a stress test once the CPU reaches maximum utilization and stops growing.

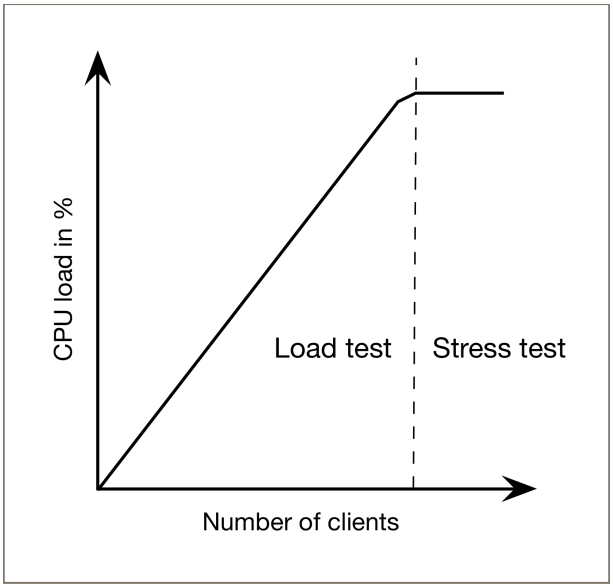

Figure 9: Relation between load and stress tests.

2.9.2 Response Times

This subsection defines the response time limits I have decided to judge my test results on. "Response Times: The 3 Important Limits" is the title of an article [31] written by Jakob Nielsen and is an excerpt from his 1993 book "Usability Engineering". In this article, Nielsen presents three response time limits for all types of applications, including web applications. The article says:

> "**0.1 second** is about the limit for having the user feel that the system is **reacting instantaneously**, meaning that no special feedback is necessary except to display the result.
> **1.0 second** is about the limit for the **user's flow of thought** to stay uninterrupted, even though the user will notice the delay. Normally, no special feedback is necessary during delays of more than 0.1 but less than 1.0 seconds, but the user does lose the feeling of operating directly on the data.
> **10 seconds** is about the limit for **keeping the user's attention** focused on the dialogue. For longer delays, users will want to perform other tasks while waiting for the computer to finish, so they should be given feedback indicating when the computer expects to be done. Feedback during the delay is especially important if the response time is likely to be highly variable, since users will then know what to expect."

When interpreting and discussing my test results in Chapter 4 and 5, these limits will be helpful to separate good results from bad results.

2.10 Test Expectations

As long as the CPU load is below a maximum, and the test stays a load test, I expect all three servers to keep the response times fairly low. When the server reaches maximum CPU load, I expect the response times to grow slowly and linearly with the client count. Figure 10 illustrates the expectation.

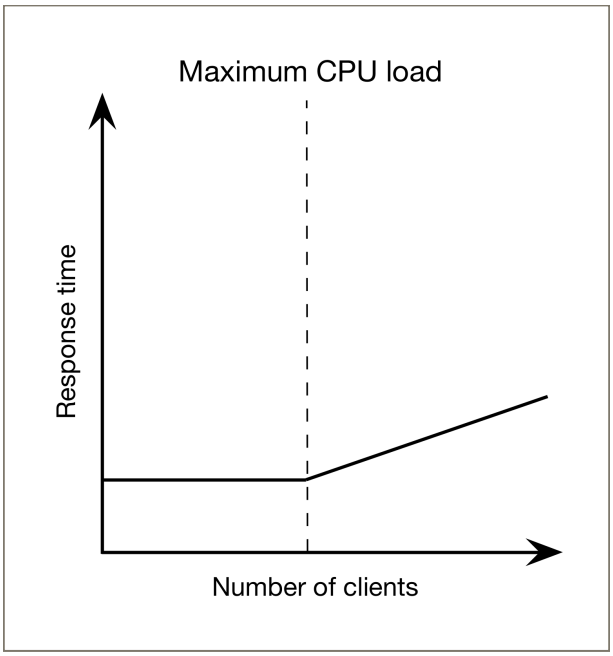

Figure 10: Expected response times.

After reading the related academic work and learning about the three different transports, I expect WebSocket to perform best. I expect Server-Sent Events to closely follow WebSocket and Long Polling to be the worst performer by far.

On the server-side, headers take time to process. Because WebSocket enables very small headers (see Subsection 2.7.4), I believe it will have a much lower response time compared to the other two transports, especially compared to Long Polling. The header processing also affects the CPU, so consequently I expect CPU load to be lower as well. I expect Server-Sent Events to perform well, but not quite on level with WebSocket. This is because WebSocket has smaller headers and is the only transport of the three that was designed from the ground up to be performant.

Another reason why I think Long Polling will perform worst is because of its response-request-oriented nature. Each request means more headers and more CPU power used.

There is, however, one area where I expect HTTP Long Polling to be the best, and that is in terms of memory consumption. Both WebSocket and Server-Sent Events introduce connections. These connections will consume more memory than incoming HTTP requests. As WebSocket is more advanced (full-duplex) than Server-Sent Events, I expect it to be the most memory consuming of the three.

Chapter 3: Methodology

The methods presented in this chapter are designed to answer the research questions from Section 1.2. The main research question is composed of three sub-questions. The first two sub-questions are related to performance while the third is related to programmer friendliness and ease of use.

To answer these questions, I have designed and implemented two test scenarios. They performance test WebSocket, Server-Sent Events and Long Polling. The test results will give answers to the performance related questions. And the gained experience of implementing the tests, will answer the programmer friendliness related one.

The thesis introduction presented two types of real-time applications. The stock price application was an example of a unidirectional messaging system while the chat application was a bidirectional messaging system. These two examples are the basis for the two test scenarios in this chapter. Each test scenario has three implementations, one powered by WebSocket, one by Server-Sent Events, and one by HTTP Long Polling.

First, I briefly introduce the two scenarios. Then I discuss what data points to collect and when to collect them. After that, I present and discuss the hardware and software platform, as well as test configurations. Then I go into a detailed description of the test scenarios and their implementations. Finally, I discuss some of the limitations of the methodology.

3.1 Test Scenarios

This section gives a quick description of the two test scenarios and the components they contain. A more detailed view of the information flow for each scenario is found in Section 3.5.

3.1.1 Test Scenario 1

Note: There are three implementations of this test scenario, one with WebSocket, one with Server-Sent Events and one with HTTP Long Polling. The text describing this scenario will not distinguish between the different versions. For detailed descriptions, see Section 3.5.

The first test scenario is a real-time message broadcasting system involving three main components; a backend, a server and a given number of clients. All the clients connect to the server, and the server connects to the backend system using a long-lived connection. The backend regularly sends messages to the server, and it is the server's job to broadcast these to all the connected clients immediately. You can think of this system as the stock price application example from earlier. In a real world scenario, the clients would be web browsers.

Figure 11 shows the main components of the first scenario.

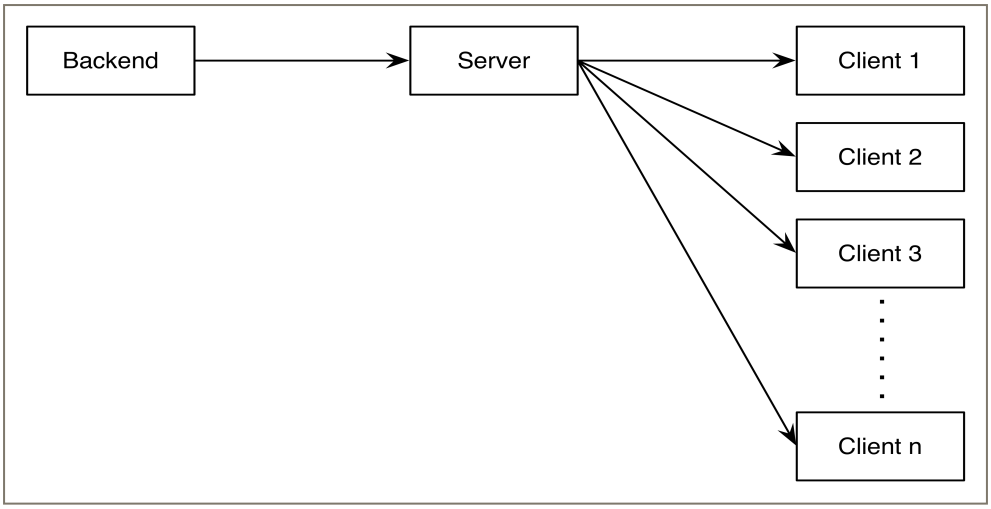

Figure 11: The three components in the first test scenario.

3.1.2 Test Scenario 2

Note: There are three implementations of this test scenario, one with WebSocket alone, one with Server-Sent Events (HTTP POST for client-to-server messages) and one with HTTP Long Polling (HTTP POST for client-to-server messages). The text describing this scenario will not distinguish between the different versions. For detailed descriptions, see Section 3.5.

The second test scenario is a real-time chat system. It consists of two main components; a server and a given number of clients. All the clients connect to the server using WebSocket, Server-Sent Events or Long Polling. During the test, each client regularly sends a chat message to the server. Each and every one of these chat messages are then broadcasted to all connected clients by the server. Figure 12 shows that Client 1 sends a chat message to the server. The server then distributes this message to all the other clients.

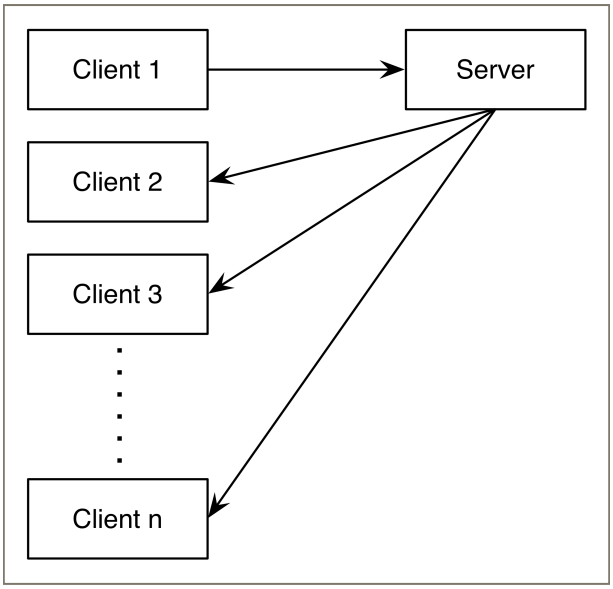

Figure 12: The components in the second test scenario.

3.2 Test Data

I have chosen to load and stress test the different transports, to get an accurate picture of how WebSocket compare to Server-Sent Events and Long Polling. Stress testing a server is done by gradually increasing the number of connected clients, to the point where the server is utilizing the CPU at a maximum. As long as the CPU utilization is below the maximum, the test is a load test.

This section describes what data is collected and how it is used to compare the three different transports on a performance level.

3.2.1 Two Points of View

There are two points of view in the tests. The first point of view is the *server-side*. From a server administrator's point of view, efficient use of server resources is important. The most interesting metrics from the server-side are *CPU load* and *memory footprint*.

The other perspective is the user's; the *client-side*. As a user of a real-time system, you do not care about how much stress the server is under, as long as the system is responsive and quick to use. The only interesting measure from a user's point of view is the *response time*. How long it takes for the system to respond to an action.

To summarize, there are three data points that are collected. The CPU load (1) and memory footprint (2) on the server-side, and the response time (3) on the client-side.

3.2.2 Collection through Three Test Phases

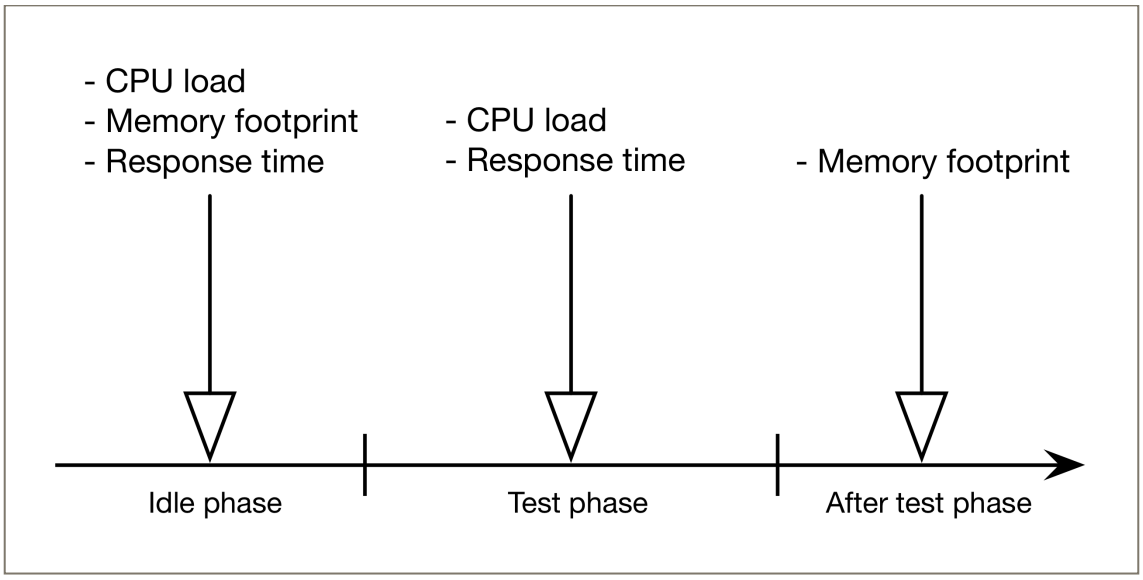

Figure 13: The three test phases and what is measured in each phase.

I have designed the tests to go through three phases. The first phase is the *idle phase*. The idle phase starts as soon as all clients are connected (polling in the case of Long Polling) to the server and the test is ready to start. In this phase, all three data points are collected.

The second phase is the *test phase*. During this phase, the test is active and running. In the first test scenario, this is the phase when the server broadcasts messages received from the backend. And in the second test scenario, this is the phase when the chat is live.

The memory footprint will rise as the test goes on and the server receives messages. But, the garbage collector can clean and free memory space as well. When this happens and how it affects memory is not easy to say. Because of this memory related uncertainty, I decided not to collect memory footprint in this phase. I can only depend on good results from the CPU load and memory footprint in this phase.

The last phase is the phase right after the test has finished running. CPU and response time will be the same in this phase as in the idle phase because the clients are inactive (or disconnected), so I will not record them. As already stated, there are uncertainties concerning memory. But I decided to collect memory footprint after the test, to make sure I spot any (if present) irregularities.

3.2.3 Collection

A separate monitoring process running independently on the server machine collects the server-side data. Every 50th millisecond this process records the CPU load and memory footprint. The server process notifies the monitoring process when the different phases start. It is the monitoring process' responsibility to calculate the CPU load and memory footprint average when the test is finished. After calculation, the average is sent to the server for print out on the screen.

The client-side metric, response time, is collected differently in the two test scenarios. In the first scenario, there is a separate ping client that every 50th millisecond sends a time stamped ping message to the server. The server immediately sends this message back, and the ping client calculates the response time. In the second scenario, each chat message going to the server includes a timestamp. And each client is responsible for calculating and recording the response time of each message it receives. An average of those recordings is calculated once the test is over.

3.2.4 Number of Test Runs

To minimize any irregular results, I ran each test 10 times for a given number of clients. An average of those ten test runs was calculated at the end.

3.3 Testing environment

3.3.1 Hardware

It is important that the server runs isolated from other components when running performance tests. The server process must not be disturbed by other part of the system, like the clients. There are several ways to isolate the server:

1. An isolated process running on the same hardware as the clients and the backend.

2. An isolated virtual machine running on the same hardware as the clients and the backend.
3. An isolated online server instances from an online cloud provider.
4. Isolated on a physical machine running in an isolated local network.

The first alternative is ideal for development as everything runs on a single computer. For testing, however, it is not. It is difficult to tell how the operating system context switches between processes and how much time it uses on the server. It would be better if the server process is the only process running, except for the operating system processes. Also, since this is about testing network protocols, it is not a good idea to run the clients and server on the same machine. And, the server should be isolated on a hardware level to get the most accurate picture of server load. As a result, option three and four remains. The two options sound both good, but I eventually landed on number four. Most online server instances share physical hardware with other instances, and it is hard to tell how the system resources are shared between them. Number four is the setup that gives me the most control over the server hardware. In addition, I had all the hardware that was needed available at home.

It is important that the server machine is considerably slower than the client machine (and backend for the first scenario). This is because the server must reach its resource limit before the client machine for this to be a stress test. Since a resource monitoring process also had to run on the server, two CPU cores or more was preferable. This way the server process could run independently on one core (Node.js is single threaded; see Subsection 3.3.2) and still be monitored without any performance hit. Of course, this all depends on how the operating system does process control, but that was the basic idea.

The server ran on the following machine:

> Apple MacBook Air 2013
> Dual Core Intel Core i5 1.3 GHz
> 8 GB DDR3
> OS X 10.10.1

The clients (and backend for the first scenario) ran on this machine:

> Apple MacBook Pro 2013
> Quad Core Intel Core i7 2.0 GHz
> 16 GB DDR3
> OS X 10.10.1

As I did not want the network to be unreliable or a bottleneck, I decided to have them both running on a cabled 1 Gb/s network.

3.3.2 Programming environment

The point of this thesis was to test and benchmark different transports (or protocols). But, benchmarking protocols is not possible as a protocol is just a set of rules. Protocol implementations, on the other hand, are possible to benchmark.

To get the most accurate picture of how WebSocket compares to Server-Sent Events and Long Polling, I would ideally compare every single implementation of the transports to each other. But that would take a very long time and is not feasible for this thesis. I followed Johannessen's advice from his thesis' Further Work section [3], where he advises to focus on a single platform.

As previously stated, Node.js is the chosen server software platform. This subsection discusses why Node.js is a good match for this thesis.

Node.js is lightweight, very performant and easy to use

When PayPal moved from a Java backend to Node.js, they saw incredible results [27]. After some tests, they could see that the Node.js server could handle double the requests per second compared to the old Java server. They say this was "interesting because our initial performance results were using a single core for the node.js application compared to five cores on Java". They also saw a 35% decrease in average response time.

In addition to being very performant, their Node.js application was "Built almost twice as fast with fewer people". It was also written with 33% fewer lines of code and 40% fewer files compared to the old Java server.

It is also worth mentioning that JavaScript is an interpreted language. This makes it feel lightweight because you do not directly compile the files yourself. Development can be fast, especially with tools like nodemon [32].

High performance and programmer friendliness makes Node.js a good fit for the one-man job this thesis is.

Node.js is single threaded

The fact that Node.js uses only one operating system process makes it perfect for monitoring. One process for the server itself and one for the monitor process can run in real parallel, as long as the CPU has more than one processing core.

PayPal wrote that their single-threaded Node.js server performed better than a five threaded Java server. That makes Node.js great at scalability: Just start another instance of the server process.

Node.js is a platform with cutting edge innovation

When looking at GitHub's most trending and starred repositories [33], Node.js is the most popular web framework by far. It is also worth noting that most of the popular GitHub repositories are JavaScript projects. Since Node.js is a JavaScript runtime, most JavaScript code written for a web browser can also be used on the backend with Node.js. That means excellent compatibility with many existing projects.

Node.js comes with the package manager NPM. NPM makes it simple to instantly fetch new pieces of code and integrate them into your system.

I only write code in JavaScript

Node.js was a breath of fresh air in the web development world when it arrived. It is not necessarily because JavaScript on the server is such a great idea. But because developers can focus on a single programming language for their entire web application, backend to frontend. I consider it a great thing only having to write JavaScript for this thesis:

- JavaScript is an expressive and dynamic programming language, meaning I can write powerful applications in few lines of code.
- It increases the readability in this thesis because there only is one programming language in the examples.
- JavaScript is everywhere. Whatever project you are working on, there is a very high probability that the project includes a web component. With the latest edition of OS X by Apple, there is even a JavaScript interface to the operating system [34]. Also, I chose it so that I can learn some of its quirks [35], as I am likely to work on a web project in the future.

Node.js is well suited for creating command line programs

Node.js has great support for creating command line utilities with the *readline* module [36]. This makes it perfect for the lightweight clients.

3.3.3 Command Line Clients

Because of similarities to Johannessen's thesis [3], I had contact with him when working with my thesis. He met challenges when using full-fledged web browsers as clients and for that reason advised me to use smaller, lightweight command line clients in my tests. Because he used web browser clients, he could not focus on scalability or stress testing. This thesis focuses solely on the transports; not frameworks like he did. I also wanted to use smaller command line clients that let me spawn them in the number of hundreds and see how the server behaves when pushed to the limit.

3.4 Test Configurations

Because the tests were designed to be load and stress tests, they had to be run in such a way that the server reached its resource limit with the chosen hardware. In this section, I will present and discuss the parameters for both test scenarios which made that possible.

3.4.1 Maximum Number of Clients

Before tweaking the test parameters, it was important to know what the maximum number of clients the client machine could handle. After some testing, it was clear that 500 had to be the maximum. As a default, OS X allowed 709 user processes. On the client machine, about 220 user processes were always running. To then spawn 500 client processes was not possible without increasing the operating system's maximum process limit. I increased the limit to 1024 with the commands in Example 16.

```
$ launchctl limit maxproc 1024
$ ulimit -u 1024
```

Example 16: How to allow 1024 user processes in OS X.

Now, 500 additional user processes was not an issue. I could have had more clients, maybe 600, but then OS X would sometimes freeze and tell me that I had too many user processes. It happened even though I was way below the limit I manually set. The only solution was a hard restart of the computer. Because of that, I decided to keep the maximum number of clients at 500.

3.4.2 Parameters Specific to the First Scenario

For the first test scenario, there were three different parameters I had to tweak. The fact that the maximum number of clients was 500 meant I had to tweak the parameters so that the tests would become stress tests at some point before all 500 clients were used. Furthermore, this had to be true for all three transporting technologies I was going to test.

How long the backend should wait between messages

Given that I could only have a maximum of 500 client processes, the backend had to quite rapidly send new messages, in order to stress the server well before reaching 500. Every *5th* milliseconds a new message is sent from the backend to the server.

The size of each broadcasted message

This parameter should resemble a real world message size, so I decided to set this to the size of a Twitter message; a tweet. The maximum length of a tweet is 140 characters. UTF-8 characters are encoded at different sizes, ranging from 1 byte for standard English characters to 4 bytes for Kanji [37]. The minimum byte size of a 140-character tweet is then 140 bytes while the maximum tweet size is 560 bytes. I decided to use a 140-English-character long tweet. Including 33 bytes of header data, each message is then of size 173 bytes.

The number of messages the backend should send per test

Each test should run long enough to minimize inaccuracies in CPU usage caused by the garbage collector. At the same time, a test should not run for too long, as that would make it unfeasible for the time I had at hand. Consequently, this number is set to 5000. 5 milliseconds between each message mean that each test runs for 25 seconds.

3.4.3 Parameters Specific to the Second Scenario

Just as with the first scenario, the 500 client limit worked as a guide for me to find the right parameter choices here as well. I wanted to reach full CPU utilization for all three transports some time before the 500 client limit.

The size of each chat message

The payload of each chat message is “Hello! How are you doing today?” That is a very short message, but it resembles a real world chat message. In addition to the payload, there are

header data, consisting of a *timestamp field*, a *from field* and a *type field*. In total 40 bytes of header data and 31 bytes of payload equals a total message size of 71 bytes.

How long each test should run

The first test was designed to run for 25 seconds. That made it run long enough for an accurate picture, but at the same time not too long and making it unfeasible to do. I chose 30 seconds for each test in the second scenario.

Message spread and frequency

Each client sends a chat message to the server every three seconds. To have an equal spread of messages, providing an even load on the server, the clients do not start sending chat messages at the same time. They are spread over the three seconds. Figure 14 shows an example of five clients sending their first two messages.

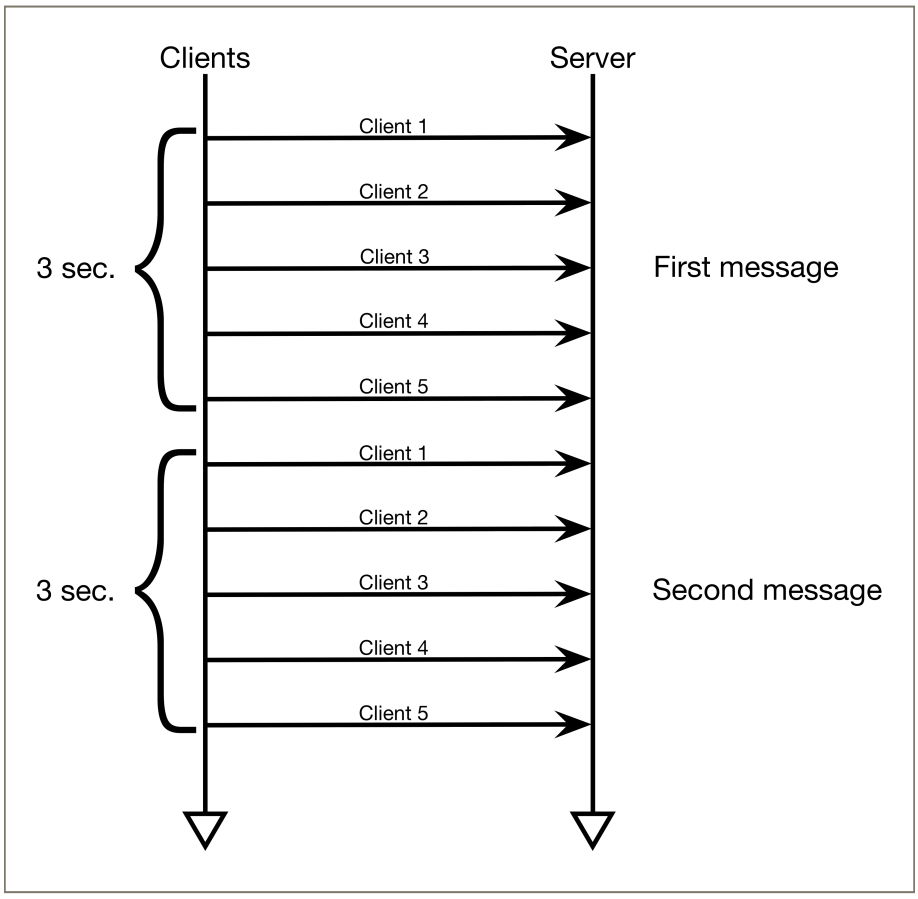

Figure 14: Example showing five clients sending their first two messages.

3.5 Detailed Information Flow

In this section, each scenario is expressed in detail both in words and by sequence diagrams. The two scenarios share two common concepts:

- Master client: As a user of the tests, you never initiate the clients themselves directly, but always through a master client. The master client is a process responsible for spawning the desired number of clients and reporting the calculated average response time.

- Monitoring process: This is a process responsible for measuring the CPU load and the memory footprint of the server. It is spawned by the server process itself.

3.5.1 Test Scenario 1

The first test scenario, the unidirectional broadcast application, has three components, a backend, a server and a given number of clients. Figure 15 shows how the information flows during a test.

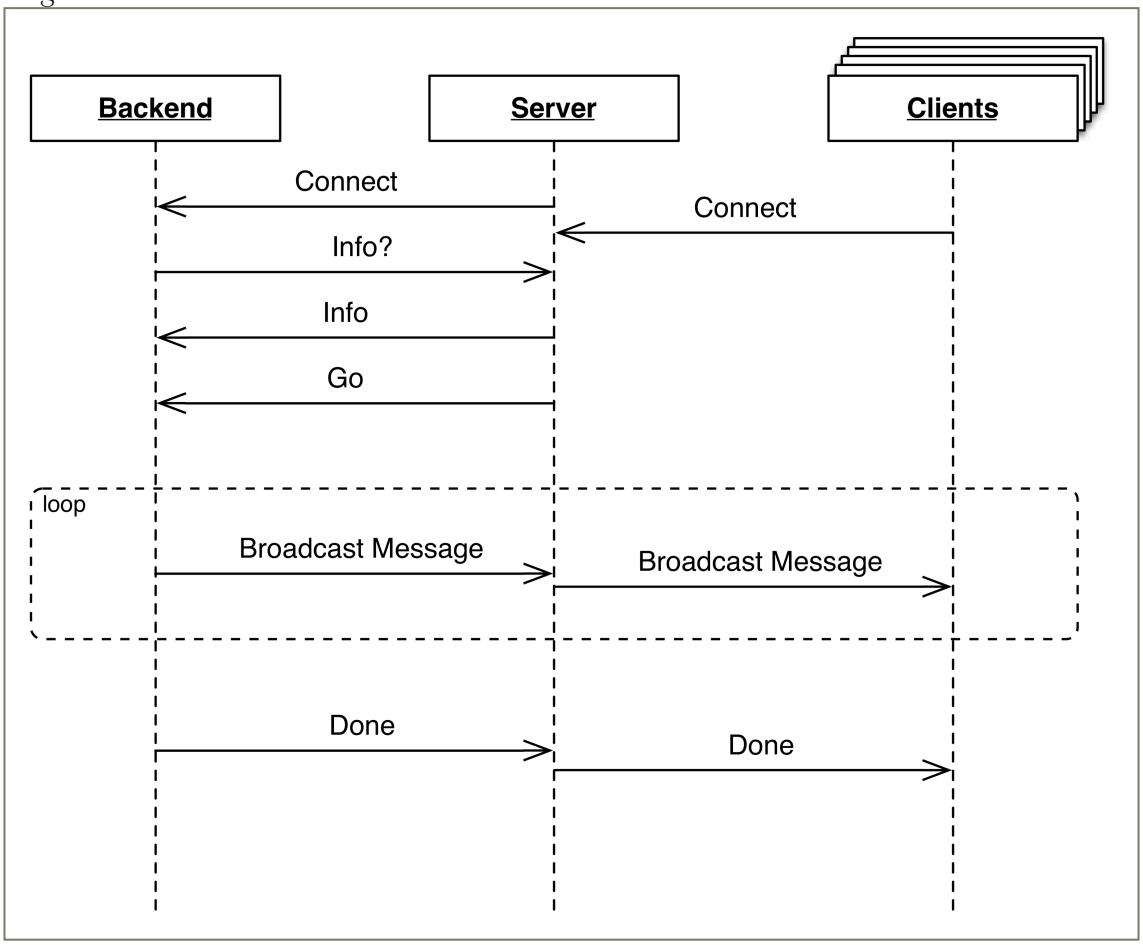

Figure 15: Sequence diagram for the first test scenario.

Server and Backend

Once the server starts, it immediately connects to the backend. The backend then sends an info message to the server, asking how often and how many times messages should be sent. The info message triggers the server to prompt the user for these parameters. Once they are typed in, they are sent to the backend and the backend awaits a *go* message to initiate the test. It is up to the user on the server to make sure all clients are connected before sending the go message to the backend. The go message is sent once the server registers a press of the return key.

Once the backend receives the go message, it sends a *getReady* message to the server indicating that the broadcast start is imminent. At this point, the server forks the monitoring process.

When the backend has sent all of its messages, it sends a *done* message, signaling the end of the test. This message is also distributed to all clients so that they are aware.

The monitoring client is also notified when the broadcast is over and calculates the average CPU and memory usage before and during the broadcast. This is sent to the server that finally prints it out to the console.

Clients

The master client immediately forks up the desired number of client processes and one ping client. The client processes then instantly connect to the server. When connected, they report to the master client. This way the master client knows when all are connected.

A client is simple; when it receives a message it just tosses it away and increments a counter to keep track of how many messages it has received. When the done message arrives, the client reports to the master client that the broadcast is finished and details whether it received all messages or not.

The ping client is a process that every 50th millisecond sends a message with a timestamp to the server. The server instantly sends this message back, and the ping client calculates the time it took to get a response. The server replies with a done message when the ping client pings the server after the broadcast is over. The ping client then calculates the average response time before and during the broadcast. This is reported to the master client for printing to the console.

3.5.2 Test Scenario 2

The second test scenario has two main components, a server and a given number of clients. Because the master client is more involved in the second scenario, it is included in the following sequence diagram in Figure 16.

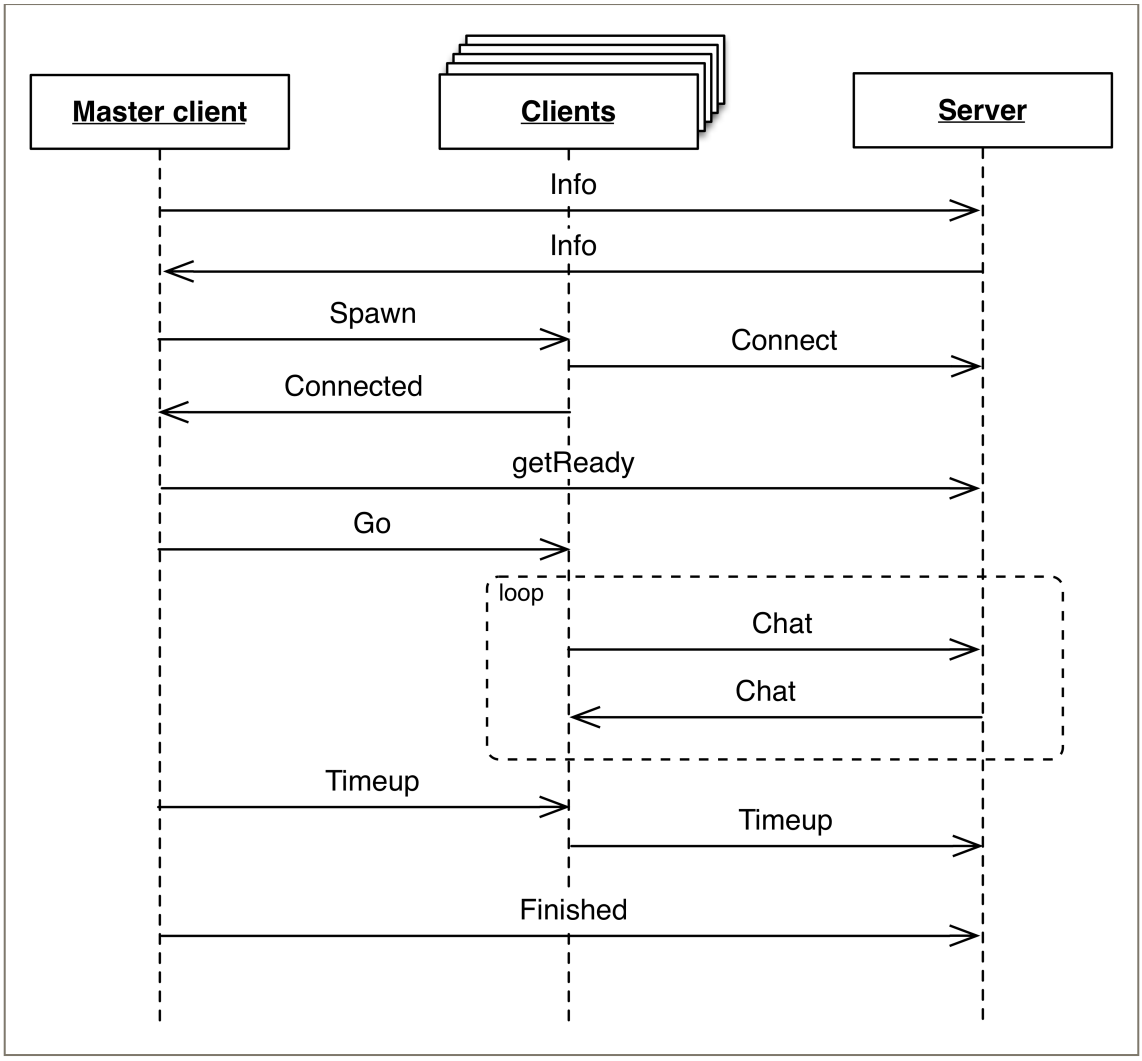

Figure 16: Sequence diagram for the second test scenario.

Clients

When the master client is started, it exchanges information with the server. This exchange makes sure that both parties know how many clients are involved and for how long the test should run. After that, the master client spawns the desired number of client processes.

Each spawned client immediately connects to the server and reports to the master client when the server connection is established. Once all clients are connected and ready, the master client sends a *getReady* message to the server. This indicates that the test is about to begin. At the same time, the master client does two things; starts a timer, and tells all clients when to start chatting with a *go* message. The timer is there to stop the chat after 30 seconds.

The clients then start to send *chat* messages to the server with three-second intervals. Details regarding the chosen test parameters are discussed in Subsection 3.4.3. Each chat message is time stamped when sent so that the clients can calculate response time themselves when they receive them.

Each client sends a *timeup* message to the server once the test timeout runs out, and the test is over. The clients then wait for a done message from the server. The done message includes the

number of messages that have been sent and the clients ensure that all messages have been received. The client then calculates response times and reports status to the master client before shutting down.

Once all clients have shut down, the master client tells the server with a *finished* message that it is safe to shut down.

Server

After the server and master client information exchange, the server waits for the getReady message indicating that the test is about start. When the getReady message arrives, the server forks a monitoring process so that it is ready to start measure server resources when the chat test is starting.

The server tells the monitor to start monitoring server resources when the first chat message has arrived. For every chat message that comes, the server immediately broadcasts it to all the connected or polling clients.

The server receives timeup messages from the clients when the chat phase is finished. The server then tells the monitor to stop collecting data. Lastly, the server waits for the master client to send a finished message. The finished message indicates that it is safe to shut down the server.

3.6 Development

This section discusses the test scenario development. There are three versions (one for each transport) of the first scenario and the three versions of the second scenario. Even though the two scenarios are different, they do share a lot of common code and libraries.

3.6.1 Common between Scenarios

As recommended by Johannessen [3], I chose to focus on a single platform, with as bare-bones implementations as possible.

With Node.js being a small JavaScript runtime and not a full-blown web framework, I had to rely on some libraries. I wanted the libraries to be as small and bare-bones as possible to have the focus on the transports and not their particular implementation. By choosing to do all tests on a single platform using small, fast libraries, and lightweight console clients, the focus could stay on the transporting technologies.

WebSocket

There are no official client or server implementations of WebSocket for Node.js, so a library had to be utilized. I could have implemented it on my own, but that would have been a thesis on its own [38]. Thankfully Node.js has a large and dedicated community, so finding WebSocket libraries was easy. Socket.IO is one example but offers way more than plain WebSockets, so that would mean a test of a library rather than a protocol. The project *ws* by Einar Otto Stangvik [39] is a server and client implementation of the WebSocket protocol for Node.js. It aims to be as close to the WebSocket API as possible. ws is also one of the fastest [40] WebSocket implementations, regardless of the platform, making it perfect for testing. In

fact, since ws is small and fast, it serves as the low-level WebSocket implementation for Socket.IO.

Server-Sent Events

There are no native server or client implementations of Server-Sent Events for Node.js. On the client-side, the choice fell on *EventSource* by Aslak Hellesøy [41]. The library is small and does not add anything on top of the technology itself.

I chose to develop the server component myself, as it is just a simple extension to a normal HTTP response. To conform to the Server-Sent Events specification, the HTTP header timeout is set to infinity and Content-Type to text/event-stream. That is essentially all that is needed for a HTTP server to become Server-Sent Events-ready. Example 17 shows how this was done in code.

```
httpServer.get('/sse', function(req, res) {
      var obj = new SSEClient(req, res);
      clients.connections.push(obj);
      req.socket.setTimeout(Infinity);

      res.writeHead(200, {
            'Content-Type': 'text/event-stream',
            'Cache-Control': 'no-cache',
            'Connection': 'keep-alive'
      });
      res.write('\n');
});
```

Example 17: Server-Sent Events Endpoint from the test code.

HTTP

There was a need for several routes into the server, and the popular web framework *Express* [42] helped to make that a reality. In addition, the small library *request* [43] made it easy to quickly send HTTP POST and GET requests from the client-side.

Resource monitoring

To monitor resource usage on the server, the Node.js package Process Monitor [44] was used. It provides a simple interface to get CPU and memory usage of a process, using the UNIX tool *ps*.

3.6.2 Software Versions

To see what version of Node.js, or any of the libraries and frameworks used in this thesis, see the Appendix under Software Versions.

3.6.3 Scenario 1 Specific Implementation Details

Backend

The backend system is essentially a WebSocket server using the same library, ws, as the server component. WebSocket was the perfect transport for the backend-to-server communication as it is fast and connection-oriented.

The WebSocket version

WebSocket is connection-oriented. That means that once the server receives broadcast messages from the backend, these can immediately be distributed to all the connected clients. This makes the WebSocket server very simple conceptually. It contains just one component for the backend communication and one for the client communication. Figure 17 shows this.

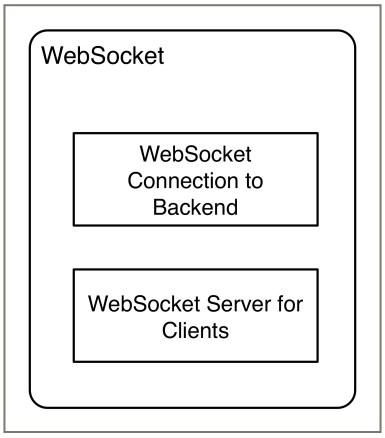

Figure 17: The WebSocket server in scenario 1.

The Server-Sent Events version

Once again, the benefits of having a connection-oriented transport make the server development an enjoyable process. The Server-Sent Events server is very similar to the WebSocket counterpart and works the same way. Figure 18 shows the architecture.

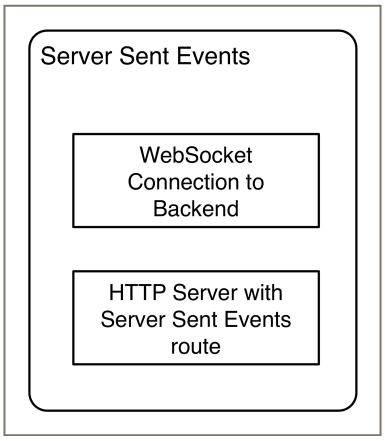

Figure 18: The Server-Sent Events server in scenario 1.

The HTTP Long Polling version

HTTP has no concept of persistent connections, so this server needed to be a bit more complex. It needs to store each message that it receives. See the problem described in Subsection 2.5.3 to understand why. Each client's polling request includes an integer to signal what message number it wants. Figure 19 shows how the server looks like with an array to store incoming messages.

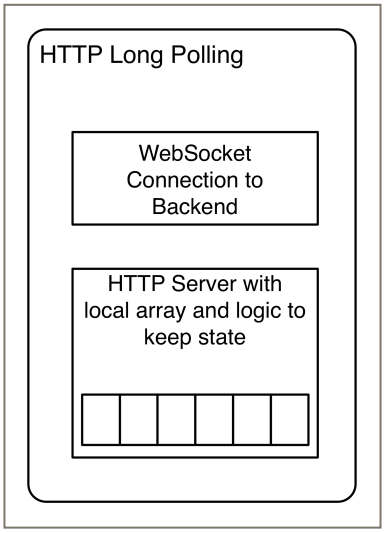

Figure 19: The HTTP Long Polling server in scenario 1.

Ping Client

The ping client is forked by the master client process. Every 50th millisecond, it sends a message to the server with a timestamp. The server immediately responds with the same message. The ping client calculates the response time when the pong is received. For the WebSocket tests, the ping client uses WebSocket. For both the Server-Sent Events and HTTP Long Polling tests, it is using standard HTTP.

3.6.4 Scenario 2 Specific Implementation Details

In contrast to the first, the second scenario has messages going server-to-client and client-to-server. Ideally this is developed using a full-duplex, stateful protocol that allows for messages going in both directions all the time. However, HTTP is not full-duplex, and Server-Sent Events is, as the name states, only for messages going server-to-client. To enable client-to-server messages, traditional HTTP POST requests were used. In a sense, the Long Polling version is Long Polling + HTTP POST, and the Server-Sent Events version is Server-Sent Events + HTTP POST. For simplicity's sake, I will only write WebSocket, Server-Sent Events, and Long Polling when I refer to the different ones.

The client spread and message frequency

As mentioned in Subsection 3.4.3, it was important to have an equal and even load on the server throughout the test. A client is programmed to send a chat message to the server every three seconds and each client is given an id number starting from 1. The process should start sending after (id * (time between each message / client count)). So client number 100 in a test with 400 clients, should start sending after (100 * (3000/400)) = 750 milliseconds.

The WebSocket version

WebSocket is an ideal protocol for this scenario as it is a full-duplex and stateful. Figure 20 shows that both incoming and outgoing messages go straight to and from the WebSocket component in the server. This is powerful and enables the server to immediately broadcast incoming chat messages. There is no need for local storing on the server-side.

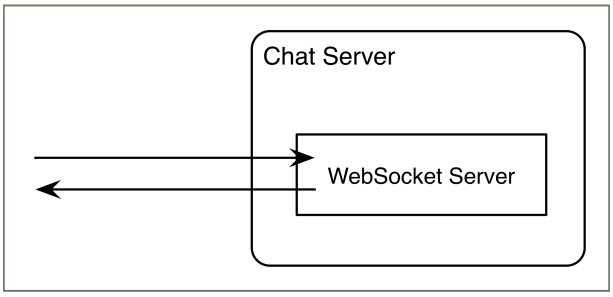

Figure 20: The WebSocket server in scenario 2.

The Server-Sent Events version

Unlike WebSocket, a Server-Sent Events server has no way to receive messages directly. An additional POST route was therefore utilized. When a client sends a chat message as a POST request, the server immediately broadcasts this message to all clients connected with Server-Sent Events. There was no need to store messages on the server-side, because the Server-Sent Events technique is connection-oriented. Figure 21 shows the server architecture.

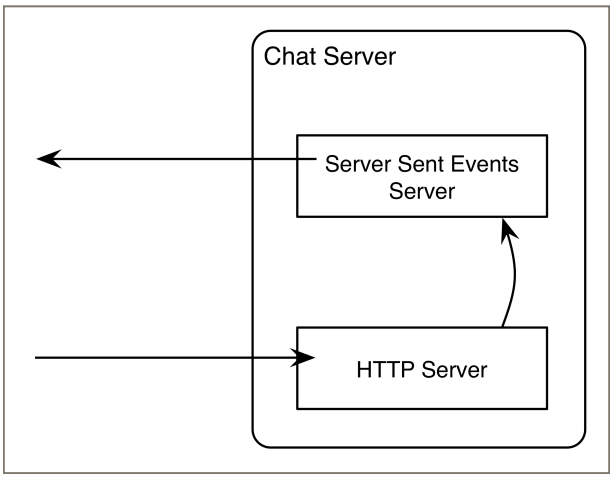

Figure 21: The Server-Sent Events server in scenario 2.

The HTTP Long Polling version

Just as with Server-Sent Events, HTTP POST had to be used for the upstream of chat messages coming from the clients. However, unlike Server-Sent Events and WebSocket, Long Polling has no concept of connected clients. Each client "loses" their connection when the polling is answered. See Subsection 2.5.3 for an example showing that there is a need to store every incoming chat message on the server-side. This leads to a significant increase in complexity. Each client's polling request includes an integer that is the index of the next message to receive. This is done similar to the system used for scenario 1. Figure 22 shows the complexity of the implementation.

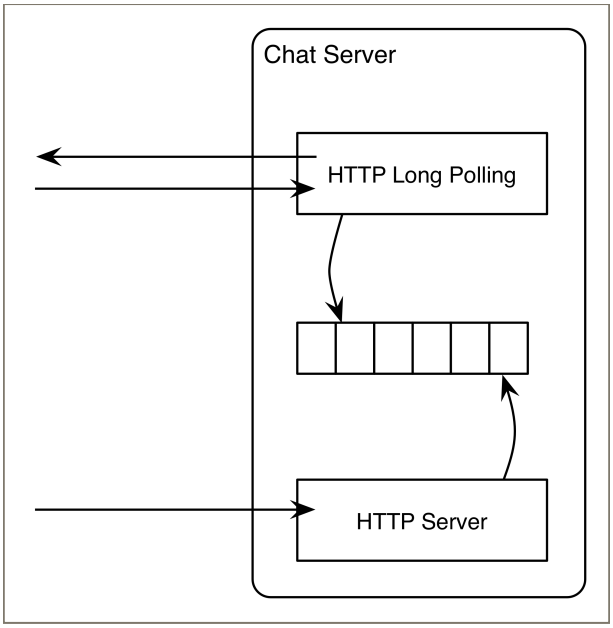

Figure 22: The HTTP Long Polling server in scenario 2.

3.7 Limitations

There are some limitations with the methodology presented in this chapter. This section looks into them one by one.

3.7.1 Performance over Longer Periods of Time

How the different transports perform over longer periods of time can also be interesting. But, for a couple of reasons, I did not measure this. First, it relies heavily on the transport's implementation. A small memory leak, for example, can compromise the results. Another reason is that performance testing over time requires very long periods of testing, possible several days for a single test. That would have been another thesis in itself, maybe on soak testing.

3.7.2 Network Use

In the background chapter, the difference in packet size between WebSocket and HTTP was pointed out. Potentially we could see a significant difference in network use (maybe even congestion) between the three transports. For simplicity's sake, the tests were designed so that the network would not be a bottleneck. That makes it easier to find the breakpoint between a load test and a stress test, as network factors can be ruled out.

3.7.3 Quality and Correctness of the Code

There is always a chance that the test code is not written in a satisfactory manner. It could even be worse, that the implementation is outright wrong. But this uncertainty will always be there, as long as humans write the code. Even with bigger projects and frameworks that are used by thousands, bugs and errors can occur [45]. I do not believe my tests are written in an incorrect or error prone way. As you will see in the next chapter, the results from scenario 1

are comparable to scenario 2. This backs up my belief that the code is written in a correct manner.

3.7.4 Only One Software Platform

The fact that I have chosen to do the performance testing on a single software platform introduces a couple of limitations to the methodology.

First, the picture I get of WebSocket, Server-Sent Events, and Long Polling is a reflection of how these transports perform on the Node.js platform, not in general. However, with the recommendation by Johannessen [3], and the fact that it is not feasible to test all platforms, performing the tests on a single platform was my choice.

The other limitation is that relying on a single platform for the tests, makes it vulnerable to errors or bugs in the chosen platform. A bug or fault in Node.js would to a certain degree compromise the results.

3.7.5 Node.js Event Loop and Garbage Collection

As discussed previous in this chapter, Node.js is a good fit for the tests in this thesis. However, there are some aspects of Node.js one needs to be aware of.

The Event Loop is a critical part of Node.js which needs to be understood. I briefly presented it in the background chapter but did not mention the following weakness. When the Event Loop has triggered and called a callback function, it is blocked. Usually not for long, but if it is stuck on a computationally demanding task, it can cause slow response times or connect issues [46]. With being an interpreted and dynamic language, JavaScript cannot be optimized in the same way as e.g. Java. This makes Node.js a great platform when each event is light, but not so great when events are computational heavy.

As stated in Subsection 2.8.2, Node.js' V8 garbage collector also introduces challenges related to unpredictable slow-downs and high response times. A consequence of this is that the memory footprint after the test can be inaccurate. Maybe one time the test runs, the garbage collector has not yet collected dead objects while another time it has. Because of these uncertainties concerning the garbage collector, I did not want to focus much on memory in my results.

Chapter 4: Test Results

This chapter presents the results from the two test scenarios. These results are the foundation for the discussion found in the next chapter that aims to answer the performance related research questions.

Each test scenario was run ten times for a given number of clients and an average of those ten runs was calculated. The client count goes from 1 to 500 with increments of 50.

Each scenario has three implementations, one for each tested transport.

For the complete test result records, see the Appendix.

As stated in Subsection 3.2.2, the tests have three different phases, as Figure 24 shows. The results in this chapter are presented using the same division:

1. Idle phase: CPU load, memory footprint and response time.
2. Test phase: CPU load and response time.
3. After test phase: Memory footprint.

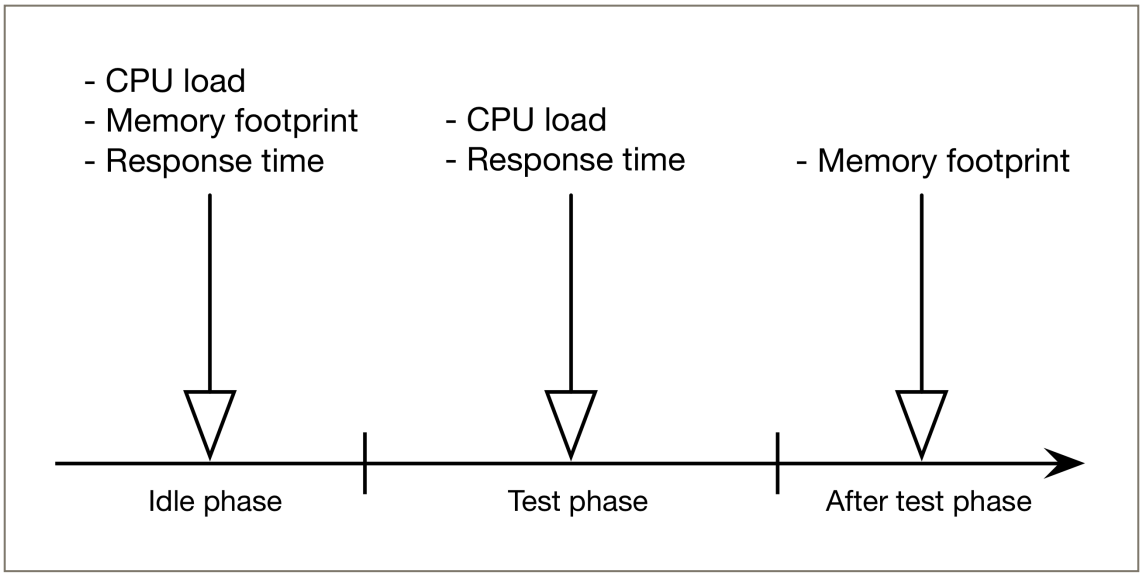

Figure 23: The three test phases and what is measured in each phase.

4.1 Idle Client Phase

As discussed in the previous chapter, the first test phase is the phase when all clients are connected/polling, but inactive and idle. Idle clients should consume as little server resources as possible enabling for short response times for active clients.

In this phase, I do not distinguish between the two test scenarios, as they behave similarly before the test phase.

4.1.1 CPU Load

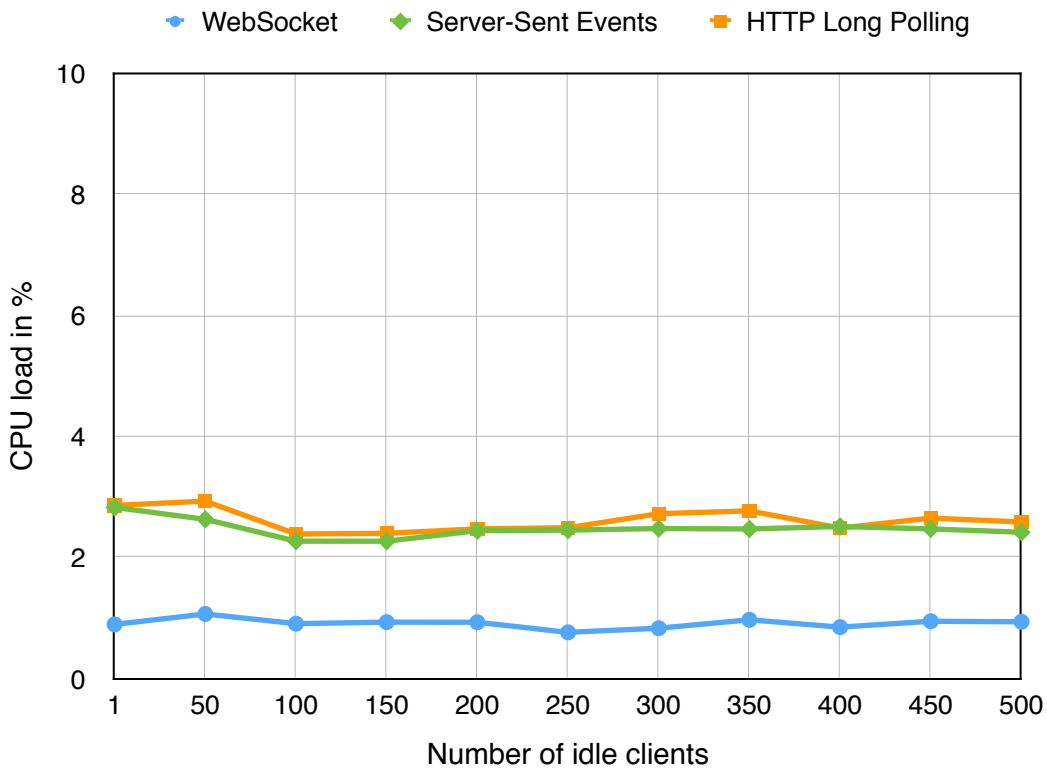

Figure 24: CPU load in the idle phase.

All three servers used so few CPU cycles that the Y-scale of Figure 24 had to be set to a maximum of 10% to set them apart. The Long Polling and Server-Sent Events servers perform very similar, almost identical, with a CPU load between 2% and 3%. The WebSocket server uses even less CPU power and stays around just 1%.

Even as the client count rises all the way up to 500, the CPU usage stays low and it seems unaffected by the massive increase in idle clients. This is true for all three servers.

The fact that the Long Polling and Server-Sent Events servers perform so similarly, can be explained by their implementation. Both servers use the same library for HTTP support, Express.

These results show that all three servers support 500 inactive clients with no issues related to CPU use. The WebSocket server is the best performer, but the results from the other two servers must also be considered good.

Section 2.10 proves that these results were expected. They were expected because idle clients are no work for the server. The only area where I expected idle clients to have an effect on the server was in memory consumption.

4.1.2 Memory Footprint

Figure 25: Memory footprint in the idle phase.

In Figure 25, we see the HTTP Long Polling and Server-Sent Events servers both start off below 30 MB. Their memory footprint then slowly rises as the client count increases. The Server-Sent Events version consumes more memory than the Long Polling variant, but not by much.

The WebSocket server starts off at a very small footprint of only 17 MB, but sees a larger growth as the client count increases. When the client count reaches 500, it has overtaken the Long Polling versions and is just barely lower than the Server-Sent Events counterpart.

The similar starting point for the Long Polling and Server-Sent Events servers can be explained by their common use of the Express library. As the client count increases, they start to differ because a Server-Sent Events connection is taking up more space than a hanging Long Polling request.

These results clearly show that a hanging HTTP Long Polling request consume less memory than a Server-Sent Events or a WebSocket connection. We can also see that a WebSocket connection is considerably more costly than a Server-Sent Events connection. Section 2.10 shows that these results were expected.

4.1.3 Response Time

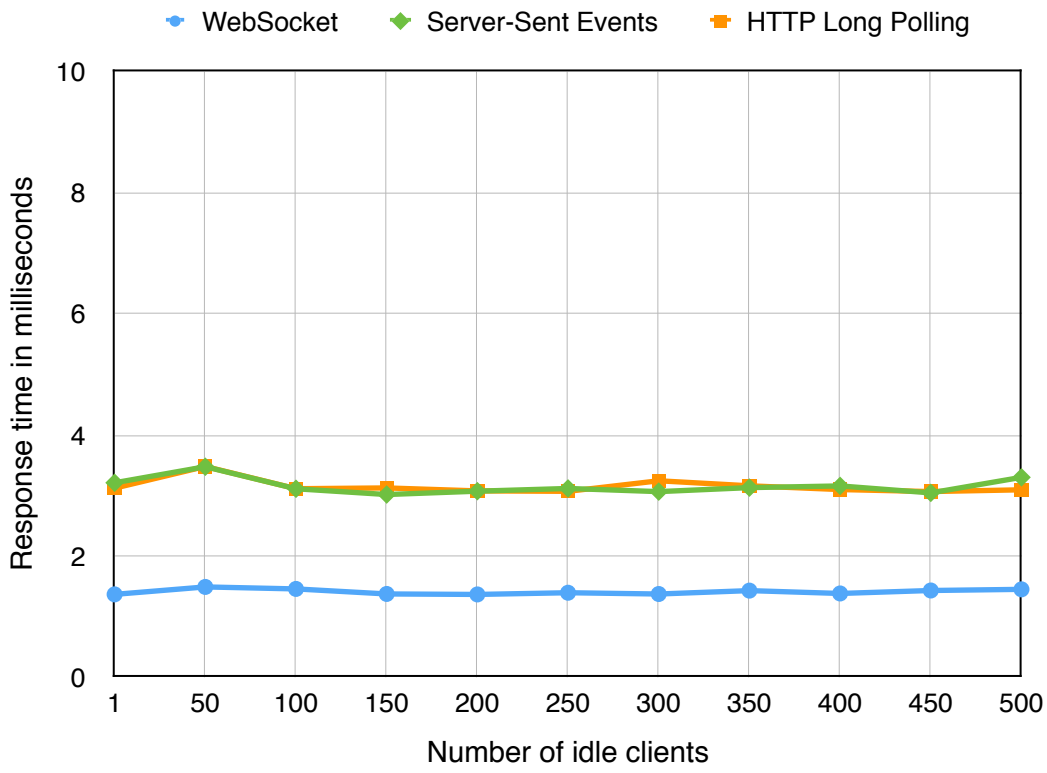

Figure 26: Response times in the idle phase.

Figure 26 shows the server response times in the idle phase. Once again, we see that the Long Polling and Server-Sent Events servers have very similar results. Both servers respond to ping messages within 3 to 4 milliseconds, well below any of the response time limits presented in Subsection 2.9.2 (1.0 and 0.1 seconds). The WebSocket server is even more impressive, with response times always below 2 milliseconds.

Similarly to the CPU load, the response time is unaffected by the increased number of clients.

The almost identical results between the Long Polling and the Server-Sent Events servers might be explained by two factors. First, they both use an HTTP ping route by the same HTTP library. Second, they are both "pinged" by the same client. The WebSocket version, on the other hand, has a WebSocket based ping client.

These results are in line with my expectations found in Section 2.10. I predicted that the WebSocket version would be the fastest, because of HTTP header processing and WebSocket being designed for performance.

4.2 Test Phase - Scenario 1

The results in this section show how the three different transports performed in the first test scenario. The data points are collected from right after the broadcast phase has started to just before it ends. This way, abnormalities from the initialization and teardown are eliminated.

The test phase is the most interesting phase as it aims to answer the performance related research questions. As discussed in Subsection 2.9.1, when the CPU load is below maximum, the test is a load test, and when it reaches a maximum and stops to grow, the test is a stress test.

4.2.1 CPU Load during Broadcast

Figure 27: CPU load in the first test scenario.

Figure 27 shows that the Long Polling server reaches maximum CPU utilization with only 50 clients. Already at that point, the server is stressed. On the other hand, the Server-Sent Events and WebSocket servers, reach maximum CPU load with 250 and 350 clients respectively.

It was expected that the Long Polling server would see a steeper climb in CPU load compared to the other two. But it was not expected that it would reach the peak CPU load this quick. Apart from that, these results are in line with the expectations from Section 2.10.

These results show that, from a server CPU load perspective, Server-Sent Events and WebSocket appear to be more suited for doing server-to-client real-time messaging than HTTP Long Polling.

4.2.2 Response Time during Broadcast

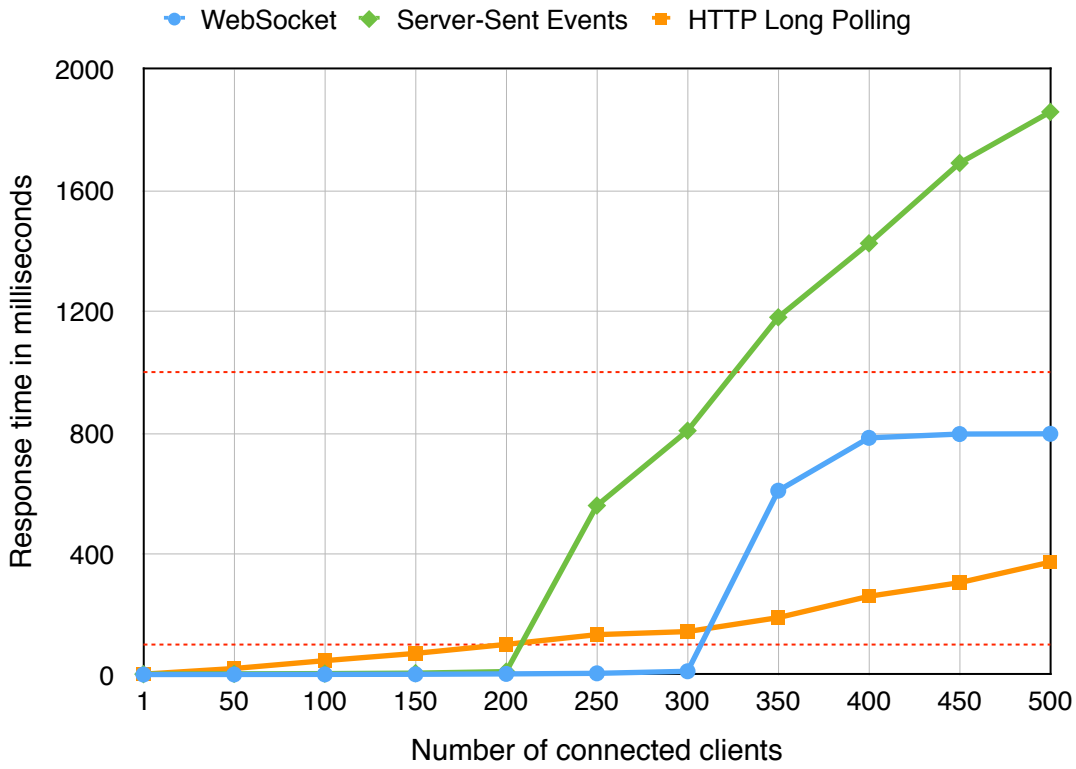

Figure 28: Response times in the first test scenario.

The two dotted red lines in Figure 28 are the 1.0 and 0.1-second limits described in Subsection 2.9.2.

When the HTTP Long Polling server reaches full CPU utilization with 50 clients, we see the start of a steady, almost linear climb in response time as the client count increases. With 200 clients, the server breaches the first red line and now uses more than 100 milliseconds to answer a request. With the maximum of 500 clients, the server responds after 372 milliseconds, well below the 1.0-second limit.

The Server-Sent Events server has a very low response time as long as the CPU load is below the maximum. From 1 to 200 clients, it always responds within 11 milliseconds, which is well clear of the 0.1-second limit. We can see that the server response time starts to increase dramatically as the CPU reaches maximum load with 250 clients. It immediately breaches the 0.1-second limit and with 350 clients, it even surpasses the 1.0-second limit. At 500 clients, the Server-Sent Events server uses over 1.8 seconds for each response. Although the response time seem to climb linearly (in line with expectations from Section 2.10), the climb is unexpected steep.

The WebSocket server shows similarities to the Server-Sent Events version. As long as the CPU load is moderate, from 1 to 300 clients, the response time stays very low. That matches my expectations from Section 2.10 about how well WebSocket suits this kind of application. With 350 clients, the test becomes a stress test as the server CPU is fully utilized. The response

time now jumps well above the 0.1-second limit. Unlike the Server-Sent Events server, the WebSocket server's response time growth stops at 800 milliseconds with 400 clients.

Once again, this sudden and explosive growth in response time is unexpected. For the response time in the first test scenario, it is only the HTTP Long Polling server that gives expected results. As soon as the servers are CPU constrained, the Server-Sent Events, and WebSocket versions struggle to keep up with the technically more complex and outdated HTTP Long Polling. These results are not only unexpected, but also surprising.

Lastly, it is interesting to see how the response time for the Long Polling server stays fairly low, even though, the CPU is stressed.

4.3 Test Phase - Scenario 2

This section presents the results from the test phase in the second test scenario, the chat system. The data points are collected from right after the chat phase is live to right before it ends. This minimizes the possibility of collecting bad results from when the test is initializing or tearing down.

4.3.1 CPU Load during Chat

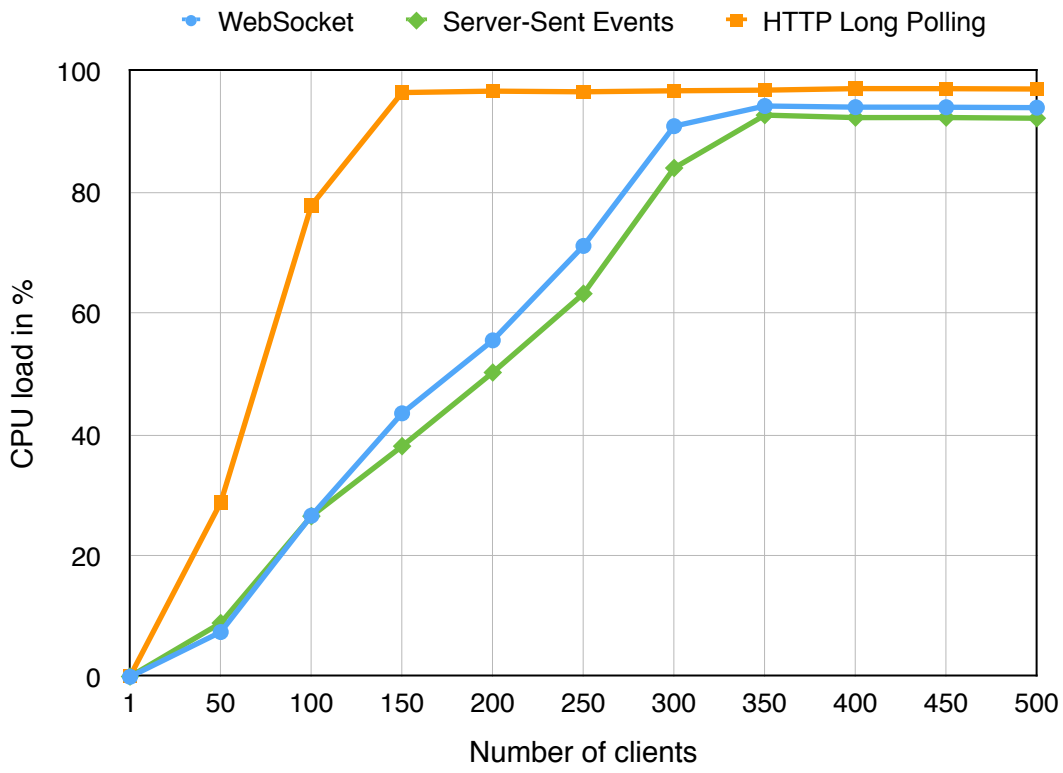

Figure 29: CPU load in the second test scenario.

Figure 29 shows that the Long Polling server reaches the maximum CPU utilization with 150 clients, while the other servers reach their maximum at 350 clients both. That Long Polling is the transport to first reach its maximum is in line with the expectations from Section 2.10.

The Server-Sent Events and WebSocket servers perform equally as good here, with Server-Sent Events slightly ahead. At this point, it does not look like the increase in header size for the Server-Sent Events server has compromised CPU performance compared to the WebSocket version.

4.3.2 Response Time during Chat

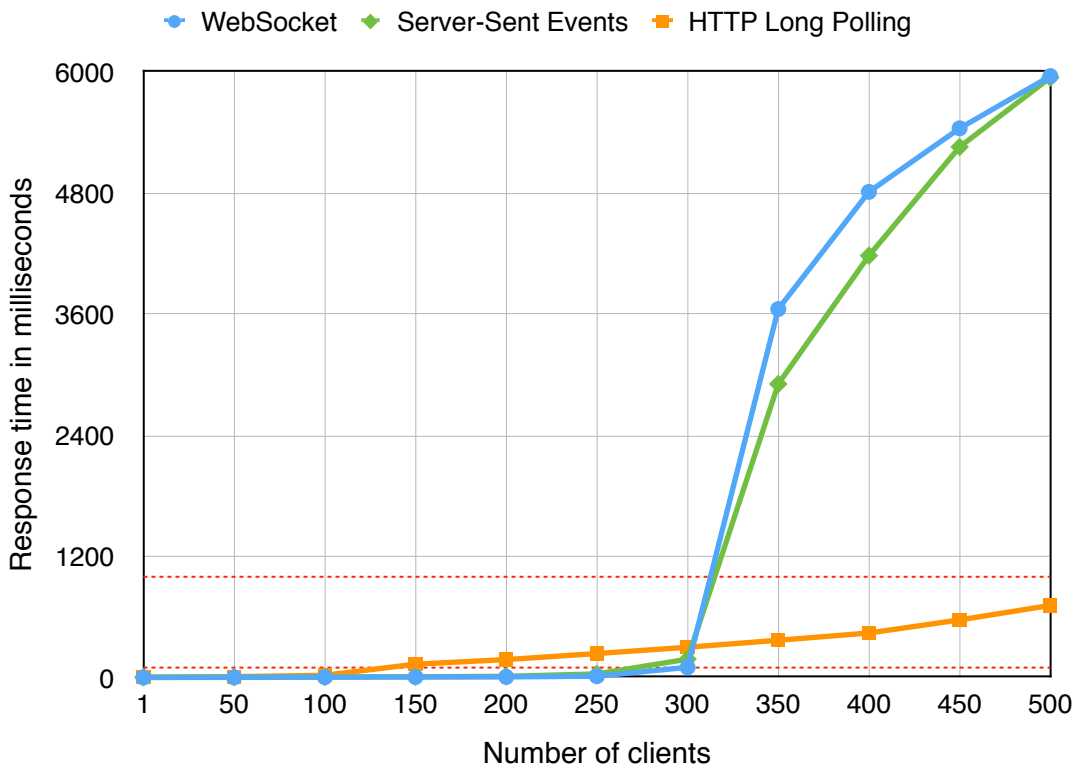

Figure 30: Response times in the second test scenario.

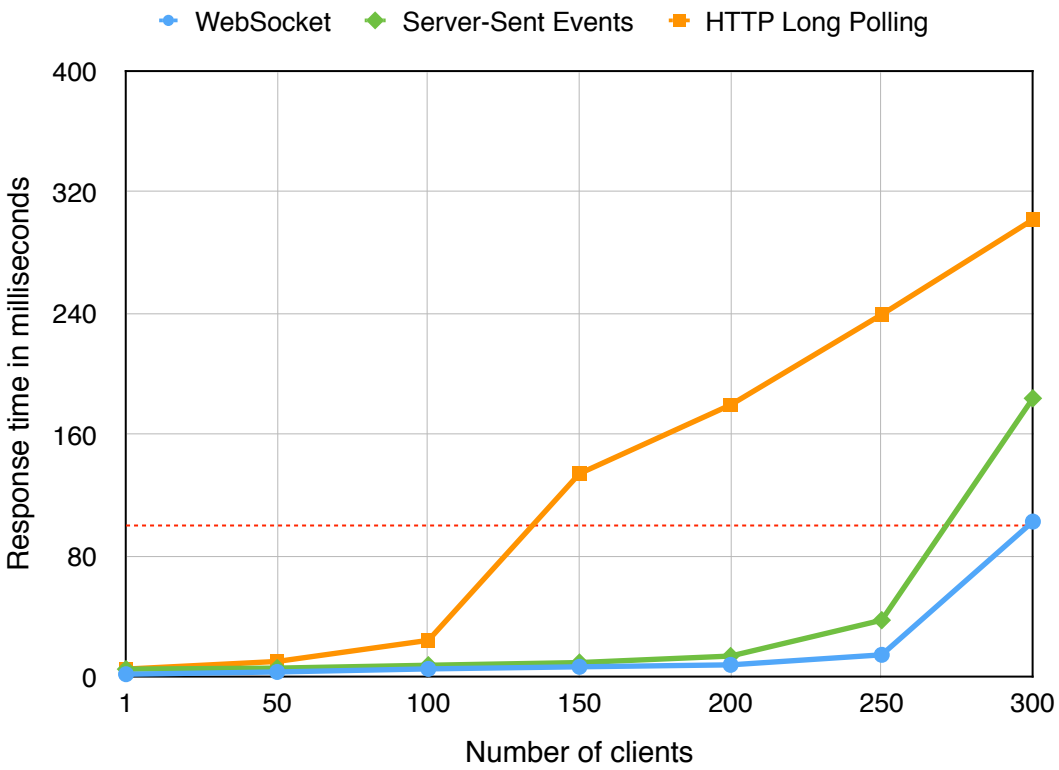

Figure 31: Zoomed in response times in the second test scenario.

Figure 30 shows the response times for the second scenario's test phase. Because the response time goes all the way up to 6 seconds, I have included Figure 31. That figure shows the same results, but zoomed in to make them easier to read before the dramatic increase in response time.

Again, the dotted red lines in the figures above indicate the 0.1 and 1.0-second limits found in Subsection 2.9.2.

When the Long Polling server reaches full CPU utilization with 150 clients, we see the start of a steady, almost linear increase in response time as the client count rises. The Long Polling server behaves as expected (see Section 2.10), and as long as the CPU load is moderate, it never uses more than 0.1 seconds to respond. Once the test is at stressing levels, can we see that the response time breaks the 0.1-second limit, but never the 1.0-second limit.

The Server-Sent Events and WebSocket servers performed similarly in this scenario. They both reach maximum CPU load with 350 clients. But already at 300 we see that the CPU is pushed hard enough for the response time to jump significantly. As long as the CPU load is moderate and only at load test levels, both servers have response times well below the 0.1-second limit.

When the load reaches a stressing level, the Server-Sent Events and WebSocket server show an explosive and sudden growth in response time. With the maximum of 500 clients, both servers have response times of 6 seconds. That is way above the 1.0-second limit.

The fact that the Long Polling server reaches full CPU utilization before the other two was expected because it is technically more demanding (see Subsection 2.7.4) in terms of headers. It was also expected that the server would have a slow linear growth in response time when the CPU is stressed.

As long as the CPU load is moderate for the Server-Sent Events and WebSocket servers, they behave as expected. But as soon as they are under stressing levels of load, they show unexpected high response times. The results were in line with the results from the first test scenario (Subsection 4.2.2), even though they were unexpected.

4.4 Memory Footprint after Tests

This section presents the recorded memory consumption right after the test phase has finished. Because of the memory related uncertainties presented in Subsection 2.8.2 and 3.7.5, I did not put much attention to memory in this thesis. But, I decided to include these recordings as they show interesting and unexpected results.

In Figure 32 and 33, you can see the memory consumption right after the tests have finished. In the first scenario (Figure 32), you see that the memory consumption explodes for the Server-Sent Events and WebSocket servers. This happens when they reach full CPU utilization, with 250 and 350 clients respectively. Both consume well over 1 GB of memory when they end at 500 clients. The expected results would be lower and along the line of the Long Polling server, which lands on 97 MB, more than 10 times lower. Section 2.10 shows that the expected result was a slow, gradual linear increase in memory consumption and not the explosive growth we see here.

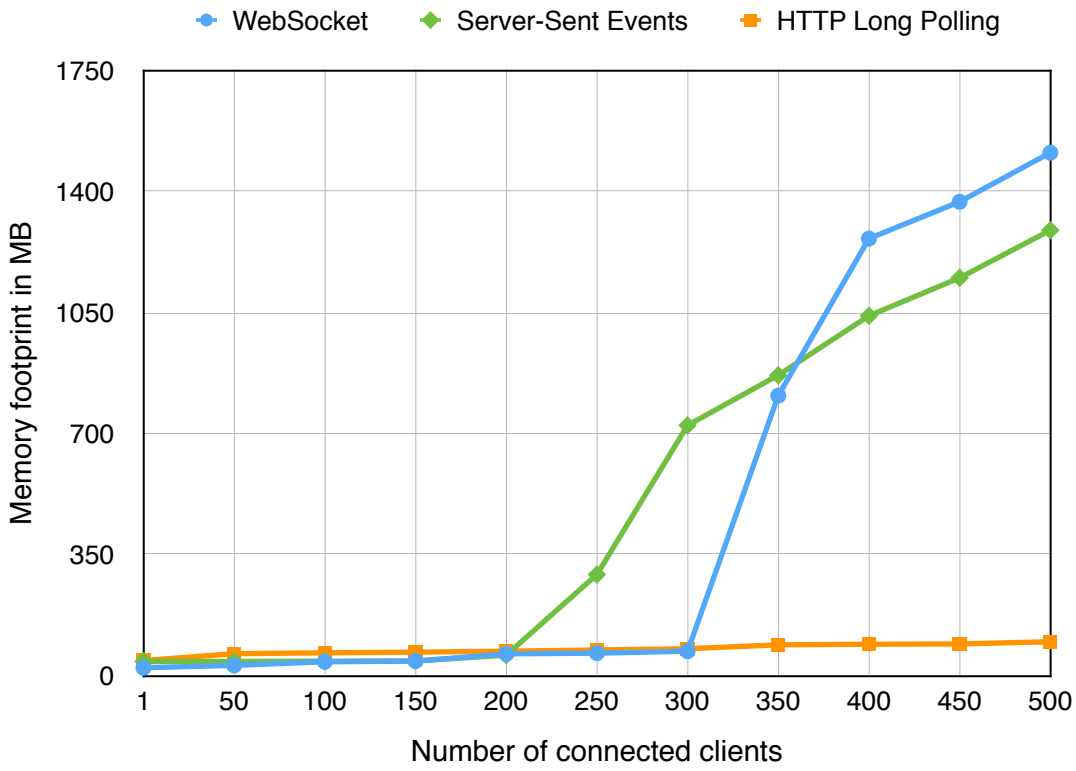

Figure 32: Memory footprint after the first test scenario.

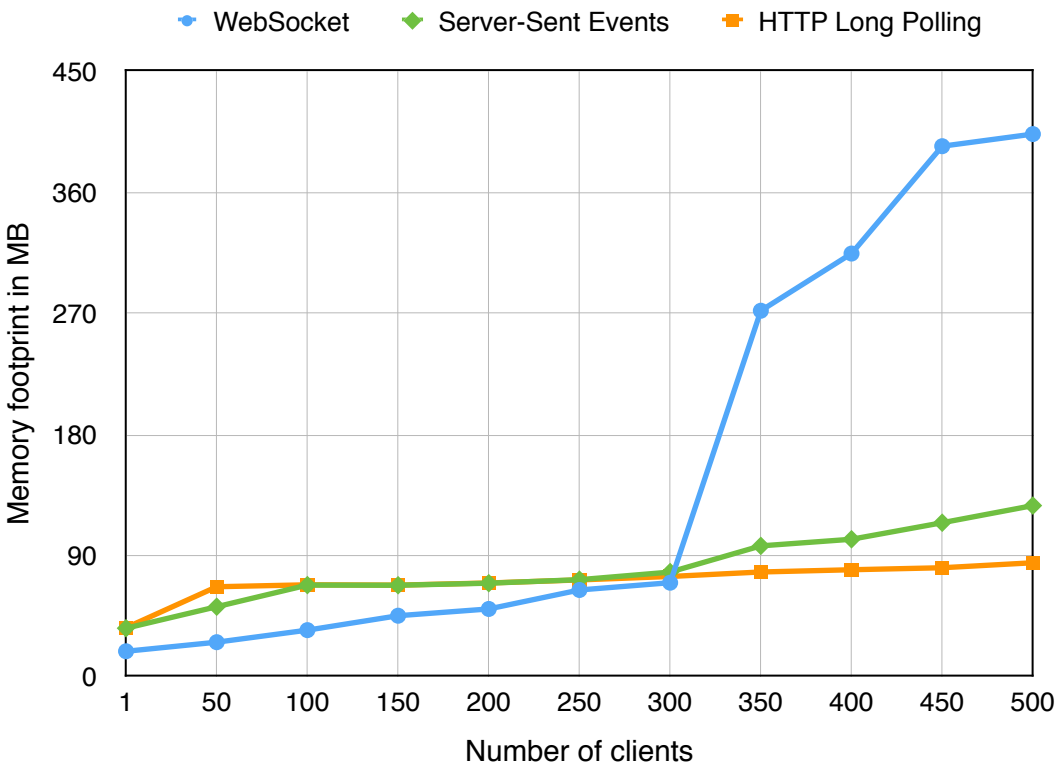

Figure 33: Memory footprint after the second test scenario.

In the second test scenario, found in Figure 33, we see the same story, although not at the same magnitude. When the Server-Sent Events and WebSocket servers reach full CPU utilization with 350 clients, there is a bump in memory usage. The WebSocket version is the most notable.

As long as the server CPU load is low or moderate, the memory consumption is exactly as expected, growing linearly with the client count. But, as the servers are under stressing load levels, it is, once again, only the Long Polling one that behaves expectedly.

4.5 Result Summary

Table 1 shows a ranking of the three transporting technologies based on their performance as servers in my two scenarios. Because of unexpected results happening under the stress tests, I have chosen to divide the ranking between the load tests and the stress tests.

During the load tests, when all servers behaved as expected (see Section 2.10), the ranking was the same between the two scenarios. The WebSocket server had the lowest response times, and it reached higher client counts before the CPU was stressed. Even though a WebSocket connection is more memory costly than a Server-Sent Events connection or a hanging Long Polling request, the difference is not that big.

The Long Polling server was the only server that behaved expectedly under stressing levels of load and constantly showed good results. The other two servers had explosive growth in both response time and memory consumption, with the WebSocket server being the worst.

These two-sided results indicate that the research questions will have two-sided answers. And the results seem to conflict with Johannessen's view that "WebSocket is better than HTTP in every aspect of real time applications." [3]

	Load test	**Stress test**
Scenario 1	1. WebSocket 2. Server-Sent Events 3. HTTP Long Polling	1. HTTP Long Polling 2. WebSocket 3. Server-Sent Events
Scenario 2	1. WebSocket 2. Server-Sent Events 3. HTTP Long Polling	1. HTTP Long Polling 2. Server-Sent Events 3. WebSocket

Table 1: Ranking of the three transports in both test scenarios.

Chapter 5: Discussion

Some of the results presented in the previous chapter were expected, while others were unexpected and surprising. Because there were unexpected results in the stress tests, this chapter divides the discussion between the load and the stress testing. The test result discussion in this chapter will be the basis for the answers to the performance related research questions in the thesis conclusion. At the end of the chapter, I will discuss how the two different scenarios were to implement from a programmer's perspective. This will let me answer the programmer friendliness related research question.

5.1 Idle Clients

Even though 500 client processes was the maximum my client test computer could handle, 500 idle clients have seemingly no effect on the server's performance. The CPU load stays as low as it was with just 1 client, and the same goes for the response time, never breaching the 0.1-second limit found in Subsection 2.9.2. This is true for all three servers, but the WebSocket version has the edge in terms of CPU usage and response time.

Another thing that the idle client phase proves is that each WebSocket connection is considerably more memory costly than a Server-Sent Events connection or a hanging Long Polling request. Even though this memory penalty is noteworthy, I do not think it is significant enough for it to be a deal-breaker when deciding whether to use WebSocket or not.

According to Section 2.10, all the points and observations from the idle client test phase results were expected.

5.2 Load Testing

In this section, I will discuss how the three transports handled low and moderate levels of load. That means all parts of the test phase where the CPU load is less than the maximum.

The Long Polling servers displayed similar results in the first and the second test scenario. They were the first to reach maximum CPU load, well before the other servers. And, they constantly used the longest time to answer ping requests.

Figure 34 and 35 show the response times from both test scenarios, but only where the tests were load tests (CPU load is less than the maximum). In the first scenario, the Long Polling server reached full CPU utilization with only 50 clients, so it was only with 1 client that it was a load test. The Server-Sent Events and WebSocket servers performed better and could handle 200 and 300 clients respectively, before reaching stressing load levels. As long as the levels of load are moderate, they all respond quickly.

In the second test scenario, we see a similar story with the Long Polling server, as it reaches full CPU utilization early on. Even though, the WebSocket and Server-Sent Events servers are

not fully utilizing the CPU before 350 clients, we already at 300 see signs of the server being CPU limited, as the response times breach the 0.1-second line.

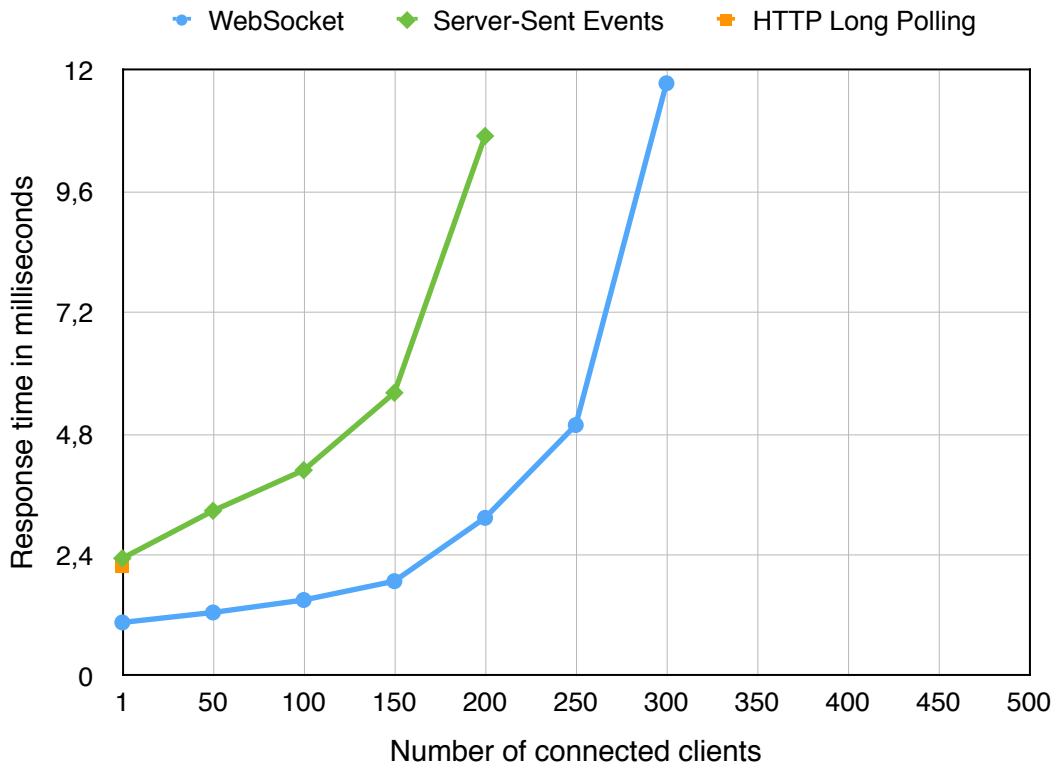

Figure 34: Response times in the first test scenario until the CPU was limited.

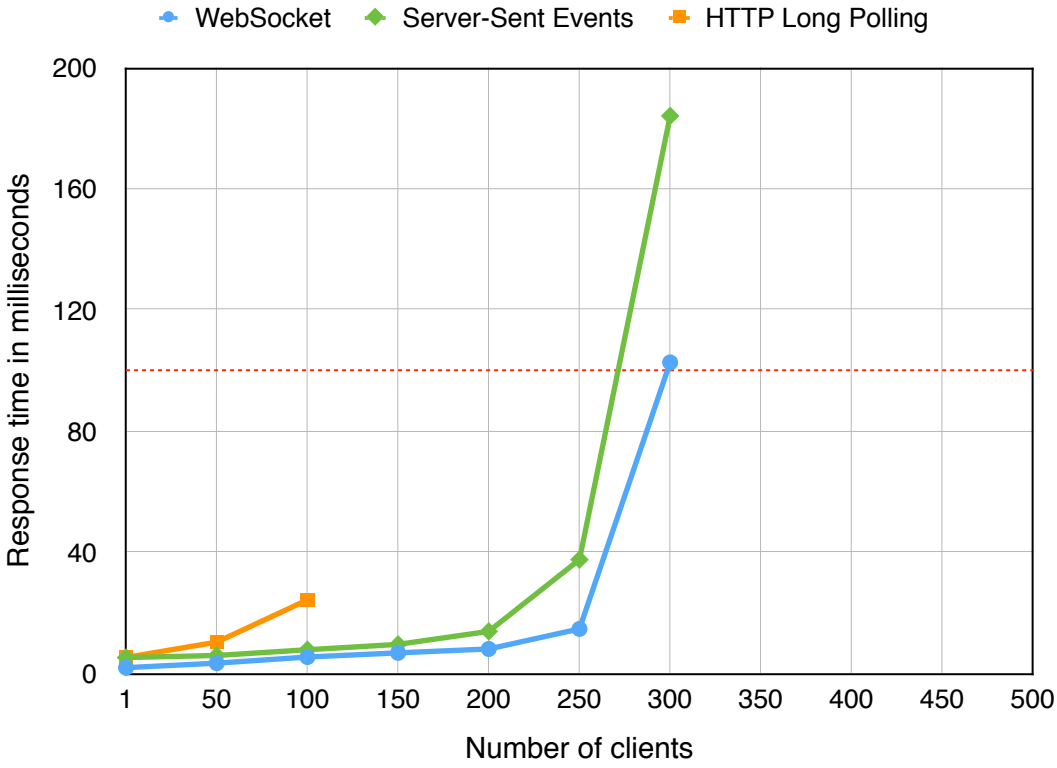

Figure 35: Response times in the second test scenario until the CPU was limited.

These results point out the following observations:

5.2.1 The Long Polling Server Performs Expectedly Bad

The HTTP Long Polling server performed poorly compared to the other two versions in the load tests, both in terms of CPU usage and response time. This was expected (see Section 2.10) and can be explained by the large amount of headers that needs processing, for both incoming requests and outgoing responses.

5.2.2 The Server-Sent Events Server Performs Great

That Server-Sent Events would perform almost as well as WebSocket in the first scenario was expected, as the transport has the concept of connections. But it was surprising that the Server-Sent Events server was so close to the WebSocket version in the second test scenario. That scenario introduced bidirectional messaging, a feature that Server-Sent Events in itself does not support. Even with the separate HTTP POST route for incoming messages, the Server-Sent Events server performed comparable to the WebSocket counterpart.

5.2.3 WebSocket is the Best Choice under Moderate Load

The WebSocket server was the most performant, both in terms of CPU load and response time. But it did introduce a small memory penalty. But that penalty is quite small, so WebSocket must be considered the best transport choice if you expect low to moderate levels of load. These results were expected, as WebSocket is a protocol designed to be a fast and thin layer on top of TCP. WebSocket seems to work great in a real-time setting.

5.3 Stress Testing

All the expectations from Section 2.10 were confirmed by the load testing. WebSocket was the performance winner, Server-Sent Events almost equally as good and Long Polling quite far behind.

The stress testing, however, did not go as expected. Figure 10 in Section 2.10 shows how the response time was expected to increase in a linear, but slow manner. In the stress tests, the Server-Sent Events and WebSocket servers show a dramatic increase in response time.

Figure 36 and 37 show the response times of both test scenarios, but only where the tests were stress tests (CPU load at the maximum). In a sense, they continue from where Figure 34 and 35 in Section 5.2 left off.

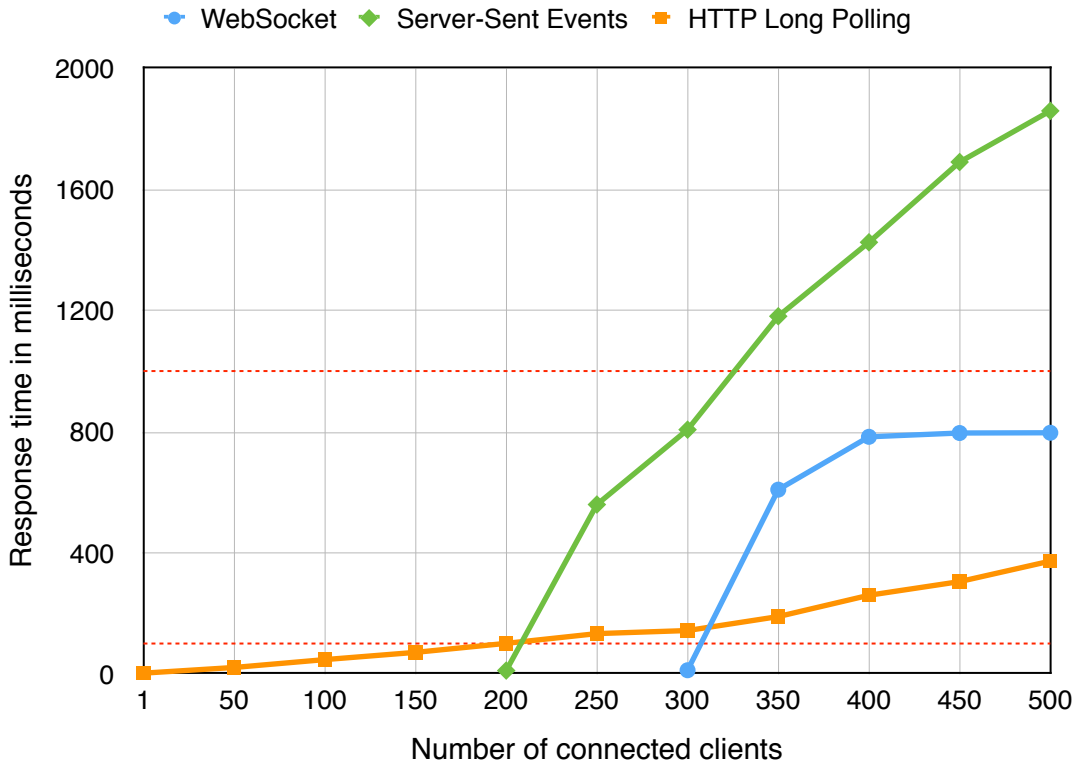

Figure 36: Response times in the first test scenario after the CPU was limited.

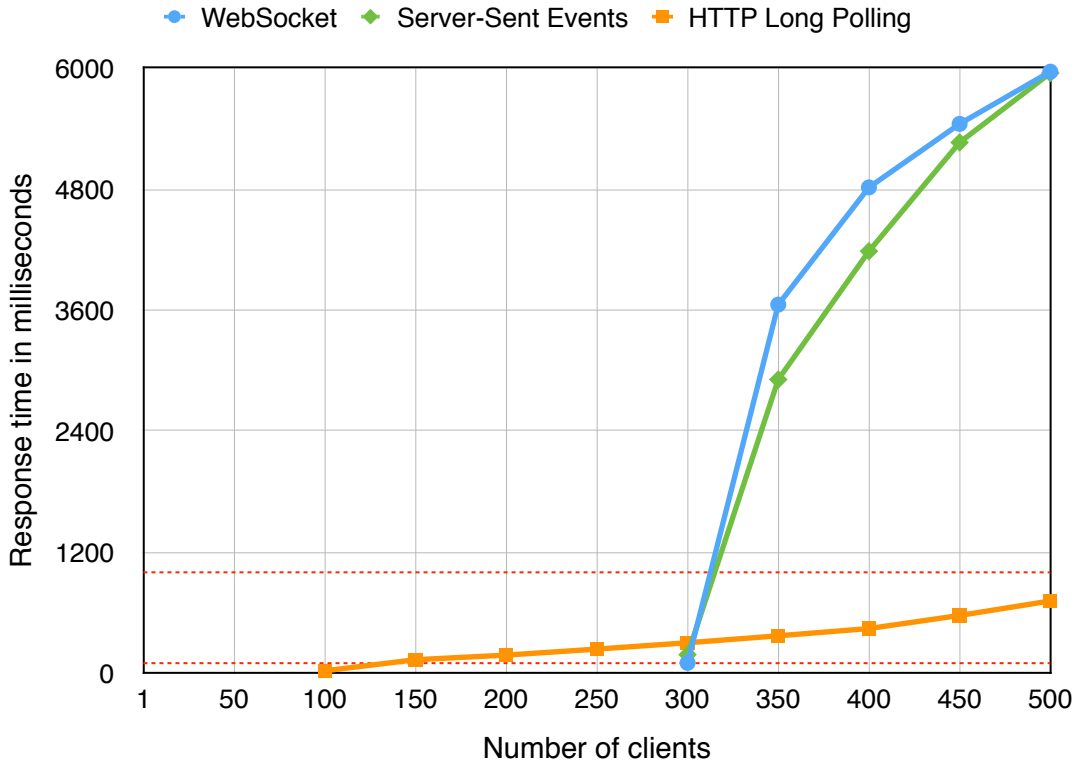

Figure 37: Response times in the second test scenario after the CPU was limited.

These results point out the following observations:

5.3.1 Unexpected and Dramatic Increase in Response Time

The linear, slow and steady increase in response time we see with the Long Polling server is the expected server behavior. The sudden spike in response times the other two servers demonstrate was unexpected and very surprising. There are several reasons why these results were unexpected:

The load tests show an entirely different picture

The Server-Sent Events and WebSocket servers performed very well in the load tests. However, they performed far worse in the stress tests, quickly breaching the 0.1 and 1.0-second limits from Subsection 2.9.2.

The HTTP Long Polling servers is fine under heavy load

In both the first and second scenario, the Long Polling servers seem to handle heavy load levels just fine and even matching my predictions and expectations from Section 2.10.

The Server-Sent Events and Long Polling servers use the same code

The Server-Sent Events and Long Polling servers both use the same HTTP library, Express. The fact that only one of them has this problem makes it very surprising.

5.3.2 Average vs. Median Response Time

The response times for the WebSocket and Server-Sent Events servers grew steep in the first scenario, but they grew even faster in the second scenario. This development was noticed early on. For the second test scenario, I consequently also recorded the median response time, not just the average. I did this to see if the average was compromised by a couple of very high response time recordings while the rest were at a lower level. The median response times were a bit lower, mostly within 75% of the average. But they were still much higher than the results from the first test scenario.

The full median and average response times can be found in the Appendix.

5.3.3 Memory

It was expected that the memory consumption would gradually and linearly increase as the client count grew. The increase would mainly come from two factors:

- The server needs memory for each connection.
- The server receives incoming messages.

The Server-Sent Events and WebSocket servers were implemented to quickly discard each received message. But each message has to be stored in memory before the garbage collector flushes it. With no easy way to inspect how the Node.js garbage collector (more explicitly, Google's V8 JavaScript engine) works, and when it runs, it was hard to tell whether there would be a significant difference between the three servers.

Because of these uncertainties regarding the memory inspection, as well as the garbage collector, I did not want memory to be a main focus for this thesis. Thankfully I did inspect

memory consumption after the tests though, as the results can point to possible explanations to why the response time explodes.

The dramatic increases in memory footprint after the tests (see Figure 38 and 39) are happening at the same time as the response times quickly escalated. In both test scenarios, the three different servers are developed using the same techniques and code styles, so there are no reasons for these sudden spikes in memory and response times to happen. The fact that the spikes in memory footprint and response time are consistently happening (see the Appendix for 10 all test runs), and the fact that they happen in both test scenarios, makes me believe there is an error, a bug or another anomaly that compromises the results.

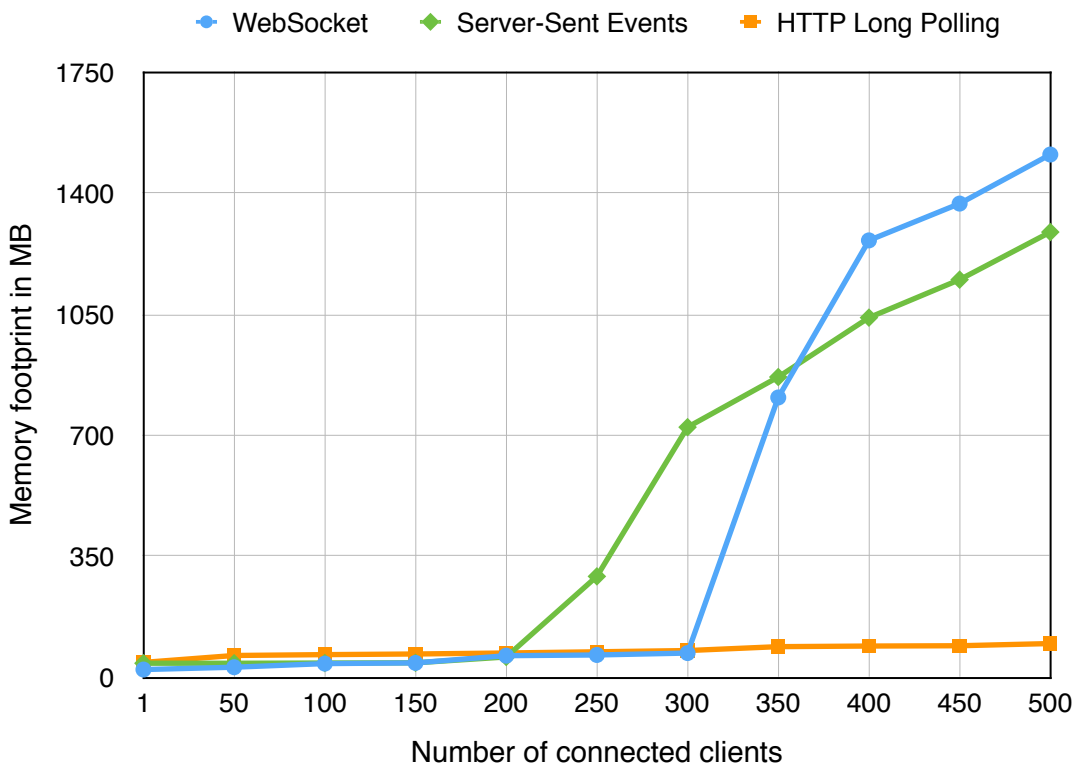

Figure 38: Memory footprint after the first test scenario.

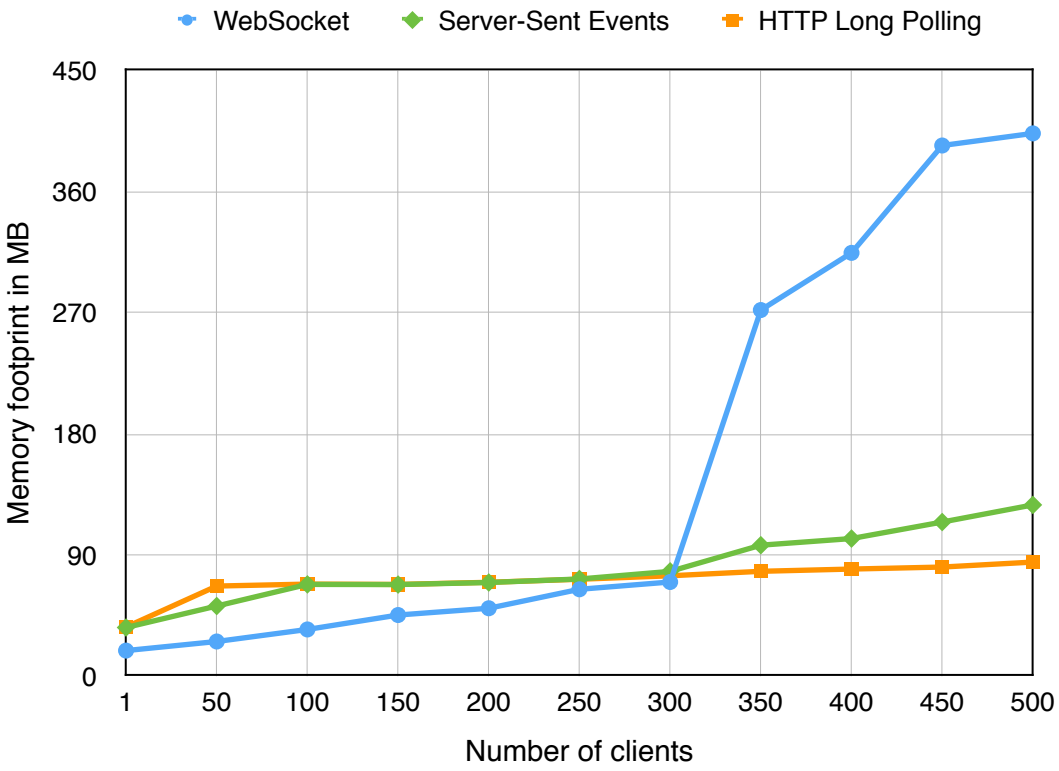

Figure 39: Memory footprint after the second test scenario.

5.4 Anomaly Discussion

This section presents and discusses possible explanations to the response time and memory anomaly.

5.4.1 Issue with the WebSocket Implementation

The WebSocket servers are affected by this anomaly. That makes it possible that there is a bug or an issue with the WebSocket library that was used. The version of ws used in the tests is *0.4.32* and as of 29th of March 2015, *0.7.1* is the latest. When looking at the change logs for version *0.5* (the version after 0.4.32) arriving November 20th, 2014, there are two very interesting changes to the library. "*Fixed a file descriptor leak*" and "*Fixed memory leak caused by EventEmitters*" [47]. Memory leaks can cause the garbage collector to become more aggressive [48], meaning increased CPU use. If these issues did occur in my tests, they could explain the high response times and large memory consumption during high load, at least for the WebSocket version.

There is, however, one important point that makes this less likely. It is not only the WebSocket servers that experience the spike in memory footprint and response time. The Server-Sent Events versions are equally affected.

5.4.2 Node.js

As stated in Subsection 3.6.1, I chose to implement the Server-Sent Events server myself. Interestingly, this means that there is no difference in libraries used by the Server-Sent Events server and the Long Polling twin (true for both test scenarios). The question of why the

Server-Sent Events version is affected and the Long Polling twin is not then arises. The answer might be linked to how Node.js treats long-lived connections. In the Long Polling versions, the connections have to be reestablished after each message sent by the server while the Server-Sent Events version keeps this connection open throughout the entire test. That is also true for the WebSocket counterpart.

There could also be a bug in the Node.js source code that only influences the Server-Sent Events and WebSocket versions. Looking for bugs or issues in the Node.js source code would take very long time and is way out of the scope for this thesis, but it is possible that there is an issue with the Node.js version used in these tests. As a consequence of being a new, innovative and fast moving platform, Node.js can suffer from bugs and instability.

Since I settled on version 0.10.35, there has happened a lot in the world of Node.js. Late last year, the open source community forked Node.js into io.js [49] after being dissatisfied by how Joyent, the organization behind Node.js, ran the project. io.js includes an updated version of the Google V8 JavaScript engine. Joyent released Node.js version 0.12 with a more updated V8 engine soon after the fork. Maybe the engines running in io.js and Node.js 0.12 fixes the test anomalies.

It is also possible that the Event Loop blocking on the broadcast function (described in Subsection 3.7.5) is a factor.

5.4.3 Node.js' HTTP Is More Tested and Stable

A possible explanation to why there could be one or more bugs in the implementation is the fact that HTTP is much more tested and in use than the other two approaches. Also, it is very rare that you push a server to the absolute limits in real world use. Maybe these abnormalities have never been seen before.

5.4.4 Errors with the Test Implementation

It is also possible that there are errors in my code. Writing bug-free code is proven to be difficult, and especially when there are few people testing the programs. For two reasons, I do not believe this is the case. First, as long as the CPU load is moderate and below maximum, nothing out of the ordinary happens. It is only when the CPU is stressed hard, that the anomalies and unexpected results occur. Second, the result anomalies happen in both test scenarios, where the implementations are different.

5.5 Implementation

Up until this point, the discussion has been related to performance and the test results. How different technologies compare in performance is very fundamental, but how easy they are to use for a programmer is also an area of great importance. In this section, I will discuss how the different servers were to implement from a programmer's perspective.

5.5.1 Test Scenario 1

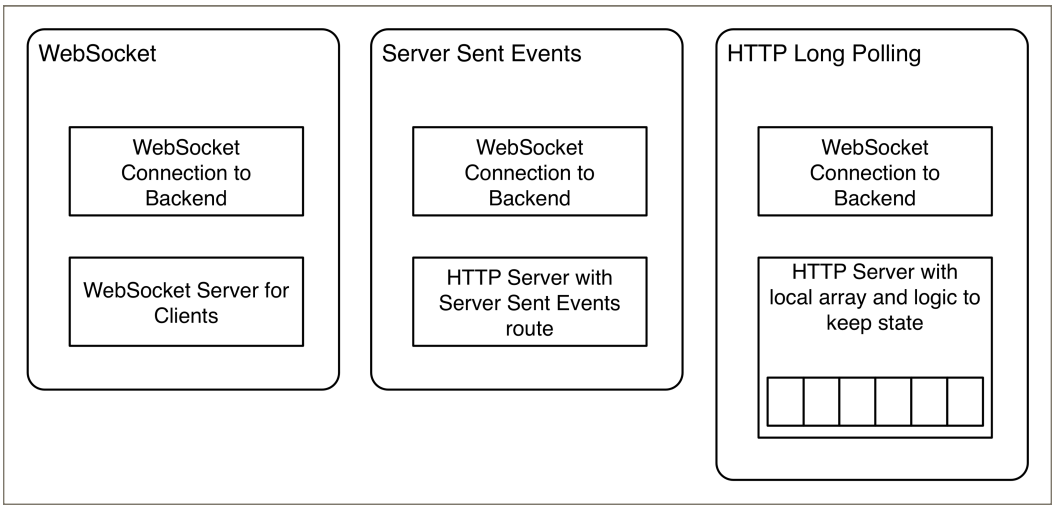

Figure 40: The three different servers in the first test scenario.

Figure 40 shows the different components involved in the three different servers for the first test scenario. Obviously they all need a WebSocket client connection to the backend server. That component is common between the three versions.

Both the Server-Sent Events and WebSocket servers were straightforward to write. The two technologies both support the concept of persistent connections, so there was no need to store incoming messages on the server; they could be broadcasted right as the server received them.

Standard HTTP, on the other hand, has no way to keep the connection open after a response is sent. That means double the network traffic and increased complexity on the server. Because of the issue presented by Figure 5 in Subsection 2.5.3, the Long Polling server must locally store each broadcast message to ensure that all clients receive them. This means a quite substantial increased complexity on the server-side, but also on the client-side. Each client must keep track of what messages it got and then tell the server what the last message it received was.

5.5.2 Test Scenario 2

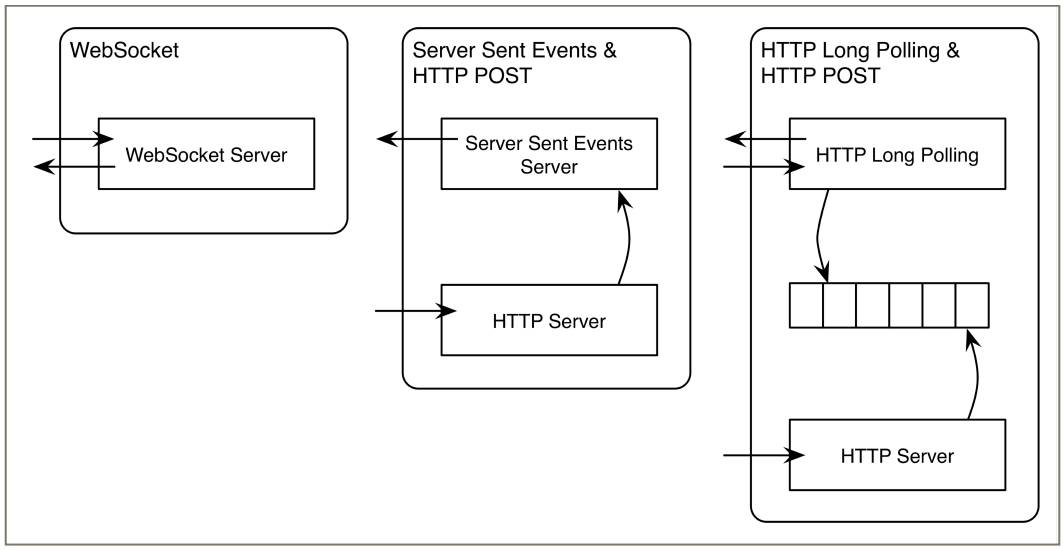

Figure 41: The three different servers in the second test scenario.

Looking at Figure 41, the difference in server complexity between the three servers becomes apparent. WebSocket is the perfect transport for the second scenario, where messages are going in both directions; server-to-client and client-to-server. As WebSocket is a full-duplex protocol, the server can be very simple, with one component for both incoming and outgoing messages. It is conceptually simple and easy to develop.

Server-Sent Events for outgoing and an additional HTTP POST route for incoming chat messages proved to be a great combination. Because Server-Sent Events allow us to keep track of connections, it was easy to distribute chat messages as soon as they were received. It was conceptually a bit more complex than the WebSocket server, but not by much.

The Long Polling server was the most complex to write. First, you need one HTTP POST route for incoming chat messages. Then, you need an additional route where the clients can poll chat messages. Lastly, as Figure 5 in Subsection 2.5.3 proves, you need a local buffer where all messages are stored (at least temporarily) to ensure that every client gets them all. This implementation was conceptually complex and more difficult to develop than the other two servers.

5.5.3 APIs

Both WebSocket and Server-Sent Events have APIs that developers can use. Long Polling, on the other hand, is a technique using HTTP and has no API available. This is huge disadvantage.

Chapter 6: Conclusion

6.1 Thesis Conclusion

In this section, I will directly answer the research questions from Section 1.2 based on the information and knowledge acquired by the test results and discussion from the previous chapters.

For simplicity's sake, when the transports are mentioned in this section, they refer to a specific combination of Node.js and libraries:

- WebSocket is the combination of Node.js version 0.10.35 and ws version 0.4.32.
- Server-Sent Events is the combination of Node.js version 0.10.35 and Express 4.9.8.
- Long Polling is the combination of Node.js version 0.10.35 and Express 4.9.8.

For a full list of all software versions, see the Appendix.

6.1.1 Sub-Questions

How does WebSocket perform compared to Long Polling and Server-Sent Events in a unidirectional, server-to-client messaging setting with high client load levels?

When the load level is high, but not extreme, WebSocket performs better than both Server-Sent Events and Long Polling. That is true both in terms of CPU usage and response time. Server-Sent Events is not far behind WebSocket, but Long Polling performed considerably worse in these tests.

However, when the server CPU load is extreme, both WebSocket and Server-Sent Events performed poorly compared to Long Polling. This is displayed by unexpectedly high response times and a large memory footprint.

How does WebSocket perform compared to Long Polling and Server-Sent Events in a bidirectional messaging setting with high client load levels?

When the load level is high, but not extreme, WebSocket performs much better than Long Polling in terms of CPU usage and response time. Despite its increase in server complexity, Server-Sent Events performs comparable to WebSocket, both in terms of CPU utilization and response time.

Also here, Long Polling outperformed WebSocket and Server-Sent Events under extreme levels of CPU load. Once again, this is displayed by unexpectedly high response times and a large memory footprint.

Does WebSocket provide any advantages over Long Polling and Server-Sent Events in a real-time setting from a programmer's perspective?

There are three major advantages that WebSocket provides over Long Polling. First is the concept of a persistent connection. The connection makes WebSocket a stateful protocol. This is especially advantageous for real-time applications where the server would like to keep information about a client over longer periods of time. The second advantage over Long Polling is WebSocket's bidirectional nature. This makes server-push a feature of the protocol and not a technique as with Long Polling. Third, WebSocket is a protocol with a well-defined API, while Long Polling is not.

From a programmer's perspective, WebSocket has two major advantages over Server-Sent Events. First, it is bidirectional by design. This makes adding a client-to-server messaging component easy. Second, WebSocket has greater web browser support.

6.1.2 Main Research Question

For what types of real-time web applications does WebSocket provide a benefit over Long Polling and Server-Sent Events?

When the server CPU load levels are below maximum, WebSocket provides both performance and programmer friendliness benefits over Long Polling and Server-Sent Events in all types of real-time applications.

However, WebSocket is a terrible choice if the CPU load levels reach a maximum. This is due to the performance penalties shown through the scenarios in this thesis. This is true from both a user's perspective with long response times and from a server's perspective with a very high memory footprint.

6.2 Further Work

The limitations presented in Section 3.7 and the test results from Chapter 4 show that there are potential for new projects as continuations of this thesis. In this section, I present suggestions to further work of this thesis as well as new and interesting areas of the real-time web.

6.2.1 Real-time Web at Scale

Even though my client machine was limited to 500 clients, that number is not that high. We can see from the Idle Client Phase results in Section 4.1 that 500 clients have no effect on server CPU load or response time. To make the server reach full CPU utilization in my tests, I had to send messages in a very rapid manner.

It would be interesting to see how the transports perform with a much higher number of clients, especially over very long periods of time. It could also be interesting to see what effect it would have to send fewer messages.

A project with this scalability focus would benefit from having several machines where client processes are running. Deploying the whole system to the cloud, using a PaaS (Platform as a Service) would make this possible.

6.2.2 More Platforms

Like I discuss in Subsection 3.7.4, the fact that I only focused on Node.js can be seen as a limitation. The picture I get of WebSocket, Server-Sent Events, and Long Polling is a reflection of how these transports perform on the Node.js platform, not in general.

It would be interesting to see how bare-bones WebSocket, Server-Sent Events, and Long Polling servers perform on different software platforms.

6.2.3 Node.js Scalability

I briefly discussed in Subsection 3.3.2 that you can scale Node.js applications by just spawning another instance of the server process. It would be interesting to see how this is done in practice with load balancing and shared server state between Node.js instances.

6.2.4 Node.js Garbage Collection

As discussed several times throughout this thesis, Node.js' V8 JavaScript engine employs garbage collection. A deep-dive into the foundations of the V8 garbage collector would be an interesting project. An interesting part of a project like this is that you can control when to run the garbage collector yourself [50].

6.2.5 HTTP 2.0

For the first time since HTTP/1.1 was standardized in 1997, we see a new version of HTTP today. Version 2.0 was just recently approved by IETF [51] and introduces several new features like header compression and server-push.

Comparing HTTP 2.0 to version 1.1 in terms of performance would be an interesting project. Maybe the disparity between HTTP and WebSocket found in this thesis looks different with HTTP 2.0.

6.2.6 WebRTC

WebRTC [52] (Web Real-Time Communication) is a new API that makes browser-to-browser communication on the Web possible. The API allows for advanced peer-to-peer communication with voice, video and file sharing without a web server. Under the hood, WebRTC is composed of many technologies and protocols that work together to make the API user-friendly.

WebRTC joins WebSocket in making the Web more advanced in terms of communication. Taking a close look at WebRTC and see what it offers can be an interesting project. Maybe it allows for entirely new applications that up until today have been unthinkable for the Web.

It could also be interesting to see how WebRTC works in combination with other protocols such as WebSocket.

References

1. H. Hämäläinen, “HTML5: WebSockets,” Aalto University, 2011.

2. V. Pimentel and B. G. Nickerson, “Communicating and Displaying Real-Time Data with WebSocket,” in IEEE Internet Computing, vol. 16, 2012, p. 45-53.

3. K. Johannessen, “Real Time Web Applications - Comparing frameworks and transport mechanisms,” Master’s thesis, University of Oslo, 2014.

4. E. Bozdag, A. Mesbah and A. v. Deursen, “A Comparison of Push and Pull Techniques for AJAX,” 9th IEEE International Workshop on Web Site Evolution, 2007, p. 15-22.

5. M. Jõhvik, “Push-based versus pull-based data transfer in AJAX applications,” Bachelor’s thesis, University of Tartu, 2011.

6. SignalR, “ASP.NET SignalR,” http://signalr.net, 2015, [Online; accessed April 29, 2015].

7. Weswit, “Lightstreamer | Global Leader in Real-Time Messaging,” http://www.lightstreamer.com/, 2015, [Online; accessed April 29, 2015].

8. Automattic, “Socket.IO,” http://socket.io, 2015, [Online; accessed April 29, 2015].

9. H. Frystyk, “The Internet Protocol Stack,” http://www.w3.org/People/Frystyk/thesis/TcpIp.html, 1994, [Online; accessed April 29, 2015].

10. I. Hickson, R. Berjon, S. Faulkner, T. Leithead, E. D. Navara, E. O’Connor and S. Pfeiffer, “HTML5,” http://www.w3.org/TR/html5/, 2014, [Online; accessed April 29, 2015].

11. T. Berners-Lee, “The Original HTTP as defined in 1991,” http://www.w3.org/Protocols/HTTP/AsImplemented.html, 1991, [Online; accessed April 29, 2015].

12. Tuffcode Ltd., “HTTP Scoop 1.4. The HTTP sniffer for Mac OS X,” http://www.tuffcode.com, 2015, [Online; accessed April 29, 2015].

13. J. J. Garrett, “Ajax: A New Approach to Web Applications,” http://www.adaptivepath.com/ideas/ajax-new-approach-web-applications/, 2005, [Online; accessed April 29, 2015].

14. A. v. Kesteren, J. Aubourg, J. Song and H. R. M. Steen, “XMLHttpRequest Level 1,” http://www.w3.org/TR/XMLHttpRequest/, 2014, [Online; accessed April 29, 2015].

15. T. Bray, “The JavaScript Object Notation (JSON) Data Interchange Format,” http://tools.ietf.org/html/rfc7159, 2014, [Online; accessed April 29, 2015].

16. A. Russel, “Comet: Low Latency Data for the Browser,” http://infrequently.org/2006/03/comet-low-latency-data-for-the-browser/, 2006, [Online; accessed April 29, 2015].

17. W3C, “Open Web Platform Milestone Achieved with HTML5 Recommendation,” http://www.w3.org/2014/10/html5-rec.html.en, 2014, [Online; accessed April 29, 2015].

18. I. Hickson, “Server-Sent Events,” http://dev.w3.org/html5/eventsource/, 2014, [Online; accessed April 29, 2015].

19. I. Grigorik, “High Performance Browser Networking,” O’Reilly, 2013.

20. V. Wang F. Salim and P. Moskovits, “The Definitive Guide to HTML5 WebSocket,” Apress, 2013.

21. I. Fette and A. Melnikov, “The WebSocket Protocol,” https://tools.ietf.org/html/rfc6455, 2011, [Online; accessed April 29, 2015].

22. IANA, “WebSocket Protocol Registries,” https://www.iana.org/assignments/websocket/websocket.xml, 2011, [Online; accessed: April 29, 2015].

23. Pivotal Software, “Spring,” https://spring.io, 2015, [Online; accessed April 29, 2015].

24. Microsoft, “ASP.NET | The ASP.NET site,” http://www.asp.net, 2015, [Online; accessed April 29, 2015].

25. R. Dahl “Node.js,” https://www.youtube.com/watch?v=EeYvFl7li9E, Talk at JSConf.eu, Berlin, 2009, [Online; accessed April 29, 2015].

26. Google, “V8 JavaScript Engine,” https://code.google.com/p/v8/, 2015, [Online; accessed April 29, 2015].

27. J. Harrell, “Node.js at PayPal,” http://www.paypal-engineering.com/2013/11/22/node-js-at-paypal/, 2013, [Online; accessed April 29, 2015].

28. npm Inc., “npm,” https://www.npmjs.com, 2015, [Online; accessed April 29, 2015].

29. Google, “Chrome V8 Design Elements,” https://developers.google.com/v8/design, 2012, [Online; accessed April 29, 2015].

30. J. D. Meier, C. Farre, P. Bansode, S. Barber and D. Rea, “Performance Testing Guidance for Web Applications,” Microsoft, 2007.

31. J. Nielsen, “Response Times: The 3 Important Limits,” http://www.nngroup.com/articles/response-times-3-important-limits/, 1993, [Online; accessed April 29, 2015].

32. R. Sharp, “nodemon reload, automatically,” http://nodemon.io, 2015, [Online; accessed April 29, 2015].

33. GitHub, “Most Starred Repositories,” https://github.com/search?utf8=✓&q=+stars%3A%3E1000, 2015, [Online; accessed April 29, 2015].

34. Apple, “JavaScript for Automation Release Notes,” https://developer.apple.com/library/mac/releasenotes/InterapplicationCommunication/RN-JavaScriptForAutomation/, 2014, [Online; accessed April 29, 2015].

35. D. Crockford, “JavaScript: The Good Parts,” O’Reilly, 2008.

36. Joyent, “Node.js Manual and Documentation - Readline,” https://nodejs.org/api/readline.html, 2015, [Online; accessed April 29, 2015].

37. T. Texin, “Unicode Supplementary Characters Test Data,” http://www.i18nguy.com/unicode/supplementary-test.html, 2010, [Online; accessed April 29, 2015].

38. N. Qveflander “Pushing real time data using HTML5 Web Sockets,” Master’s thesis, Umeå University, 2010.

39. E. O. Stangvik, “ws: a node.js websocket implementation,” http://einaros.github.io/ws/, 2015, [Online; accessed April 29, 2015].

40. E. O. Stangvik, “websocket client benchmark,” http://einaros.github.io/ws/benchmarks.html, 2015, [Online; accessed April 29, 2015].

41. A. Hellesøy, “GitHub: EventSource,” https://github.com/aslakhellesoy/eventsource-node, 2015, [Online; accessed April 29, 2015].

42. StrongLoop, “Express,” http://expressjs.com, 2015, [Online; accessed April 29, 2015].

43. Request, “GitHub: Request - Simplified HTTP client,” https://github.com/request/request, 2015, [Online; accessed April 29, 2015].

44. Modulus, “GitHub: process-monitor,” https://github.com/onmodulus/process-monitor, 2013, [Online; accessed April 29, 2015].

45. Codenomicon, “The Hearthbleed Bug,” http://heartbleed.com, 2014, [Online; accessed April 29, 2015].

46. StrongLoop, “Node.js Performance Tip of the Week: Event Loop Monitoring,” https://strongloop.com/strongblog/node-js-performance-event-loop-monitoring/, 2014, [Online; accessed April 29, 2015].

47. A. Kazemier, “GitHub: Changes in ws 0.5,” https://github.com/websockets/ws/commit/d242d2b8ddaa32f7f8a9c61abe74615767a91db4#diff-e1bbd4f15e3b63427b4261e05b948ea8, 2014, [Online; accessed April 29, 2015].

48. Mozilla, “Tracking Down Memory Leaks in Node.js - A Node.JS Holiday Season,” https://hacks.mozilla.org/2012/11/tracking-down-memory-leaks-in-node-js-a-node-js-holiday-season/, 2012, [Online; accessed April 29, 2015].

49. io.js, “JavaScript I/O,” https://iojs.org/en/index.html, 2014, [Online; accessed April 29, 2015].

50. S. McManus, “Forcing Garbage Collection with Node.js and V8,” https://simonmcmanus.wordpress.com/2013/01/03/forcing-garbage-collection-with-node-js-and-v8/, 2013, [Online; accessed April 29, 2015].

51. J. Arkko, “HTTP/2 Approved,” http://www.ietf.org/blog/2015/02/http2-approved/, 2015, [Online; accessed April 29, 2015].

52. A. Bergkvist, D. C. Burnett, C. Jennings and A. Narayanan, “WebRTC 1.0: Real-time Communication Between Browsers,” http://www.w3.org/TR/webrtc/, 2015, [Online; accessed April 29, 2015].

Appendix

Code

All the test code, as well as digital versions of the thesis can be downloaded from my GitHub repository. Direct link: http://www.github.com/oyvindrt/thesis

Software Versions

OS X	10.10.1
Node.js	0.10.35
Express	4.9.8
ws	0.4.32
EventSource	0.1.3
request	2.40.0
process-monitor	0.3.0
body-parser	1.9.0
multer	0.1.7

Table 2: The software versions used in this thesis

How to Run the Tests

Node.js is required to run the tests. Depending of the operation system default, it might also be required to increase the maximum limit for user processes.

Scenario 1

It is required to start the backend before the server. Once started, the backend listens on port 9000. The servers always listen on port 8000.

First, start the backend like this:
```
$ node backend.js
```

Then start the desired server like so:
```
$ node <ws/sse/http>server.js <backend ip> <backend port>
```
Example:
```
$ node wsserver.js localhost 9000
```

Lastly, start the clients:
```
$ node start<ws/sse/http>clients.js <server ip> <server port> <client number>
```
Example:
```
$ node startwsclients.js localhost 8000 128
```

Scenario 2

First start the server like this:
```
$ node <ws/sse/http>server.js <backend ip> <seconds the test should run>
```
Example:
```
$ node wsserver.js localhost 30
```

Then start up the clients like so:
```
$ node start<ws/sse/http>clients.js <server ip> <client number>
```
Example:
```
$ node startwsclients.js localhost 128
```

Full Test Results

Idle Clients Phase

HTTP Long Polling - Idle CPU Load

# clients	1	50	100	150	200	250	300	350	400	450	500
Run 1	3,00	2,42	2,29	2,30	2,54	2,44	2,70	2,78	2,36	2,67	2,54
Run 2	2,77	3,00	2,38	2,35	2,32	2,31	2,76	2,73	2,73	2,57	2,57
Run 3	2,81	2,97	2,41	2,42	2,41	2,38	2,64	2,93	2,43	2,62	2,77
Run 4	2,75	3,00	2,44	2,36	2,65	2,48	2,66	2,82	2,49	2,66	2,44
Run 5	2,80	3,18	2,38	2,43	2,34	2,48	2,81	2,73	2,45	2,55	2,75
Run 6	2,96	2,86	2,34	2,33	2,50	2,56	2,68	2,79	2,54	2,71	2,58
Run 7	3,05	3,00	2,44	2,30	2,55	2,57	2,60	2,50	2,54	3,00	2,49
Run 8	3,02	2,86	2,32	2,55	2,58	2,52	2,88	2,55	2,59	2,67	2,57
Run 9	2,47	3,14	2,47	2,33	2,40	2,65	2,84	3,02	2,41	2,47	2,66
Run 10	2,98	2,89	2,42	2,63	2,45	2,53	2,67	2,86	2,33	2,60	2,50
Average	**2,861**	**2,932**	**2,389**	**2,4**	**2,474**	**2,492**	**2,724**	**2,771**	**2,487**	**2,652**	**2,587**

Server Sent Events - Idle CPU Load

# clients	1	50	100	150	200	250	300	350	400	450	500
Run 1	2,71	2,88	2,37	2,39	2,35	2,40	2,32	2,52	2,79	2,52	2,44
Run 2	2,44	2,67	2,29	2,41	2,70	2,47	2,31	2,40	2,29	2,31	2,49
Run 3	2,94	2,65	2,21	2,19	2,58	2,78	2,47	2,55	2,29	2,45	2,33
Run 4	2,83	2,63	2,27	2,21	2,32	2,49	2,64	2,45	2,40	2,21	2,54
Run 5	2,93	2,49	2,26	2,26	2,56	2,44	2,53	2,53	2,72	2,42	2,32
Run 6	2,86	2,51	2,23	2,25	2,61	2,37	2,71	2,45	2,37	2,63	2,32
Run 7	3,07	2,64	2,22	2,28	2,37	2,23	2,47	2,33	2,59	2,79	2,42
Run 8	2,77	2,50	2,24	2,29	2,29	2,36	2,53	2,67	2,47	2,46	2,54
Run 9	2,80	2,66	2,25	2,24	2,25	2,49	2,62	2,40	2,74	2,54	2,34
Run 10	2,93	2,68	2,37	2,18	2,41	2,50	2,19	2,42	2,47	2,40	2,44
Average	**2,828**	**2,631**	**2,271**	**2,27**	**2,444**	**2,453**	**2,479**	**2,472**	**2,513**	**2,473**	**2,418**

WebSocket - Idle CPU Load

# clients	1	50	100	150	200	250	300	350	400	450	500
Run 1	0,98	0,65	0,48	1,04	0,91	0,83	1,02	0,97	0,89	0,72	0,93
Run 2	1,09	1,07	1,02	0,70	1,00	0,79	0,61	1,05	0,64	1,17	0,61
Run 3	1,05	1,21	0,93	0,86	1,06	0,90	1,02	0,54	1,05	1,11	1,05
Run 4	0,59	1,05	0,67	0,71	1,03	0,53	1,02	0,61	0,89	0,98	1,32
Run 5	0,62	1,02	1,09	1,15	0,93	0,88	0,51	1,05	0,81	1,07	0,73
Run 6	1,12	1,26	0,97	1,16	0,97	0,89	0,96	1,19	0,59	1,20	1,23
Run 7	1,18	1,14	1,09	1,02	0,60	0,62	1,11	1,05	0,89	0,77	0,84
Run 8	0,78	1,21	0,83	0,67	0,89	0,83	0,59	0,95	0,95	0,66	1,05
Run 9	0,59	0,95	0,88	0,49	0,98	0,83	0,82	1,09	0,82	0,95	1,16
Run 10	0,96	1,12	1,12	1,53	0,94	0,55	0,67	1,22	0,98	0,85	0,47
Average	**0,896**	**1,068**	**0,908**	**0,933**	**0,931**	**0,765**	**0,833**	**0,972**	**0,851**	**0,948**	**0,939**

HTTP Long Polling - Idle Memory Footprint

# clients	1	50	100	150	200	250	300	350	400	450	500
Run 1	28,47	29,63	30,15	30,38	30,47	30,71	30,74	31,19	31,41	31,43	33,30
Run 2	28,83	29,47	30,00	30,28	30,57	30,66	30,90	31,04	31,16	31,78	32,47
Run 3	28,62	29,46	30,07	30,21	30,43	29,88	30,81	31,04	31,16	31,53	32,08
Run 4	29,07	29,40	30,14	30,25	29,83	30,70	30,72	31,07	31,24	31,49	32,49
Run 5	28,81	29,44	30,04	30,16	30,58	30,67	30,87	31,13	31,27	31,48	32,52
Run 6	27,90	29,86	30,00	30,32	30,39	30,46	30,72	31,10	31,25	31,34	32,16
Run 7	28,73	29,80	29,93	30,18	30,50	30,77	30,99	31,19	31,23	32,26	32,38
Run 8	28,23	29,80	30,19	29,46	30,55	30,66	30,39	30,99	31,13	31,54	32,14
Run 9	27,10	29,49	30,13	30,35	30,45	30,41	30,87	31,13	31,36	32,54	32,40
Run 10	28,16	29,51	30,04	29,37	30,35	29,95	30,75	31,11	31,25	32,18	32,34
Average	**28,392**	**29,586**	**30,069**	**30,096**	**30,412**	**30,487**	**30,776**	**31,099**	**31,246**	**31,757**	**32,428**

Server Sent Events - Idle Memory Footprint

# clients	1	50	100	150	200	250	300	350	400	450	500
Run 1	29,08	28,58	29,10	29,69	31,03	31,90	32,13	31,96	33,09	34,11	34,93
Run 2	27,51	28,73	30,47	30,80	30,95	31,10	31,33	31,77	33,11	34,03	33,77
Run 3	28,60	28,37	30,12	30,78	30,43	30,71	31,33	32,36	33,15	34,58	34,99
Run 4	29,12	28,34	30,19	29,26	30,99	31,29	31,40	31,92	32,97	34,01	34,98
Run 5	28,78	30,00	30,68	30,72	29,99	31,30	31,71	31,93	33,45	33,55	34,71
Run 6	28,71	30,10	28,98	30,83	29,91	31,02	31,55	31,89	32,99	33,67	34,87
Run 7	28,88	30,01	30,24	30,65	30,96	31,20	31,45	32,08	32,75	34,55	34,87
Run 8	29,16	28,49	30,48	30,72	30,82	31,29	31,42	31,79	33,26	33,70	35,02
Run 9	28,94	28,44	30,49	29,20	30,95	31,16	31,27	31,98	37,23	34,48	34,53
Run 10	28,72	29,84	28,71	30,58	30,97	31,30	31,36	32,83	32,94	34,22	35,13
Average	**28,75**	**29,09**	**29,946**	**30,323**	**30,7**	**31,227**	**31,495**	**32,051**	**33,494**	**34,09**	**34,78**

WebSocket - Idle Memory Footprint

# clients	1	50	100	150	200	250	300	350	400	450	500
Run 1	17,76	21,15	22,89	23,50	24,05	27,08	28,70	29,89	32,25	32,37	34,24
Run 2	17,51	20,89	22,99	23,31	24,02	25,76	28,05	30,00	32,13	32,56	34,77
Run 3	17,59	20,93	22,41	23,53	23,84	25,80	28,09	29,91	32,15	32,49	34,10
Run 4	17,68	21,13	23,00	23,42	24,46	25,67	28,26	29,94	32,32	32,59	34,53
Run 5	17,82	21,23	23,08	23,51	24,04	25,86	28,31	29,93	32,06	32,52	34,46
Run 6	17,60	21,14	23,18	23,51	23,82	25,97	28,05	30,42	32,25	32,53	34,08
Run 7	17,98	20,92	22,80	23,56	24,05	25,67	28,16	29,92	32,14	32,56	34,47
Run 8	17,86	20,89	22,91	23,54	23,89	25,71	28,16	29,80	31,96	32,45	34,40
Run 9	17,86	21,22	22,72	23,50	23,89	25,73	28,11	30,05	32,31	32,46	34,25
Run 10	17,85	20,93	22,73	23,46	23,85	25,89	28,24	30,05	32,11	32,53	34,36
Average	**17,751**	**21,043**	**22,871**	**23,484**	**23,991**	**25,914**	**28,213**	**29,991**	**32,168**	**32,506**	**34,366**

HTTP Long Polling - Idle Response Time

# clients	1	50	100	150	200	250	300	350	400	450	500
Run 1	3,26	3,11	2,96	3,14	3,15	3,15	3,98	3,00	2,96	3,08	3,10
Run 2	2,95	3,27	3,30	3,62	3,12	2,92	3,24	3,09	3,11	3,04	2,91
Run 3	3,05	3,30	3,20	3,12	3,02	3,15	3,19	3,16	3,04	2,98	3,14
Run 4	3,10	3,30	3,29	3,14	3,07	3,01	3,17	3,18	3,01	3,01	2,94
Run 5	3,08	3,40	2,98	3,12	3,00	3,02	3,15	3,08	3,14	3,10	3,10
Run 6	3,26	2,97	3,19	3,00	3,15	3,13	3,05	3,31	3,09	3,09	3,42
Run 7	2,97	3,84	3,25	2,97	3,00	3,01	3,13	3,27	3,20	3,15	3,07
Run 8	3,10	3,30	2,88	3,08	3,12	3,03	3,20	3,05	3,20	3,12	3,03
Run 9	3,06	3,34	3,13	2,93	3,15	3,05	3,07	3,11	3,10	3,04	2,97
Run 10	3,33	4,91	2,89	3,08	2,93	3,17	3,20	3,30	3,08	2,99	3,21
Average	**3,116**	**3,474**	**3,107**	**3,12**	**3,071**	**3,064**	**3,238**	**3,155**	**3,093**	**3,06**	**3,089**

Server Sent Events - Idle Response Time

# clients	1	50	100	150	200	250	300	350	400	450	500
Run 1	3,20	3,36	3,18	2,91	2,92	3,39	2,85	3,21	4,05	2,95	3,07
Run 2	3,10	3,07	3,01	3,12	3,10	2,96	3,14	3,14	3,09	2,98	3,08
Run 3	3,17	3,32	3,10	2,95	3,13	3,15	2,90	3,16	3,07	2,88	3,21
Run 4	3,30	4,01	3,11	3,02	3,07	3,04	3,22	3,04	2,89	3,18	3,41
Run 5	3,32	3,02	3,15	2,98	3,11	3,06	3,18	3,10	3,04	3,01	3,56
Run 6	3,17	4,03	3,01	2,95	3,03	2,99	3,03	3,14	3,15	3,07	3,03
Run 7	3,17	3,19	3,19	3,04	3,00	3,06	3,02	3,05	3,08	3,18	3,14
Run 8	3,09	3,11	3,04	3,04	3,16	3,10	2,99	3,24	3,02	3,09	4,22
Run 9	3,27	4,35	3,13	3,07	2,99	3,17	3,11	3,07	3,03	2,98	3,07
Run 10	3,26	3,24	3,14	3,03	3,15	3,19	3,15	3,10	3,11	3,06	3,16
Average	**3,205**	**3,47**	**3,106**	**3,011**	**3,066**	**3,111**	**3,059**	**3,125**	**3,153**	**3,038**	**3,295**

WebSocket - Idle Response Time

# clients	1	50	100	150	200	250	300	350	400	450	500
Run 1	1,32	1,40	1,44	1,35	1,38	1,49	1,48	1,41	1,42	1,48	1,41
Run 2	1,46	1,52	1,40	1,32	1,47	1,32	1,34	1,41	1,39	1,45	1,42
Run 3	1,27	1,50	1,39	1,46	1,38	1,34	1,39	1,40	1,31	1,46	1,41
Run 4	1,31	1,41	1,53	1,35	1,15	1,44	1,39	1,38	1,45	1,33	1,45
Run 5	1,40	1,35	1,35	1,37	1,18	1,45	1,33	1,37	1,28	1,37	1,47
Run 6	1,37	1,46	1,45	1,40	1,40	1,37	1,28	1,46	1,38	1,48	1,55
Run 7	1,46	1,49	1,44	1,30	1,40	1,38	1,38	1,43	1,38	1,42	1,32
Run 8	1,29	1,55	1,41	1,47	1,38	1,38	1,36	1,48	1,43	1,40	1,49
Run 9	1,41	1,59	1,54	1,27	1,52	1,35	1,33	1,39	1,33	1,45	1,58
Run 10	1,33	1,58	1,57	1,40	1,35	1,37	1,40	1,51	1,40	1,42	1,35
Average	**1,362**	**1,485**	**1,452**	**1,369**	**1,361**	**1,389**	**1,368**	**1,424**	**1,377**	**1,426**	**1,445**

Test Phase - Scenario 1

HTTP Long Polling - CPU Load in Test Scenario 1

# clients	1	50	100	150	200	250	300	350	400	450	500
Run 1	17,54	98,41	98,32	98,33	98,16	97,87	97,85	97,96	98,23	98,03	98,20
Run 2	17,47	98,47	98,35	98,28	98,20	98,06	97,94	98,06	98,04	98,16	98,06
Run 3	17,73	98,35	98,38	98,23	98,03	97,98	98,00	98,05	98,01	98,26	98,12
Run 4	17,46	98,43	98,47	98,06	97,95	97,96	97,97	98,01	98,33	98,11	98,08
Run 5	17,30	98,37	98,32	98,40	98,16	98,02	97,82	98,08	98,17	98,08	98,17
Run 6	17,27	98,44	98,32	98,33	98,10	97,96	98,09	98,18	98,35	98,14	98,25
Run 7	17,37	98,30	98,46	98,35	97,97	97,80	97,93	98,00	98,00	98,27	98,21
Run 8	17,52	98,48	98,30	98,35	98,15	97,89	97,91	98,19	98,18	98,62	98,09
Run 9	17,50	98,51	98,40	98,30	98,19	97,95	98,06	98,12	98,22	98,05	98,02
Run 10	17,46	98,44	98,38	98,19	97,97	97,86	97,79	97,98	98,12	98,32	97,90
Average	**17,462**	**98,42**	**98,37**	**98,282**	**98,088**	**97,935**	**97,936**	**98,063**	**98,165**	**98,204**	**98,11**

Server Sent Events - CPU Load in Test Scenario 1

# clients	1	50	100	150	200	250	300	350	400	450	500
Run 1	8,55	32,12	57,71	79,50	93,97	97,97	97,59	98,84	99,03	99,05	99,16
Run 2	8,74	30,61	54,94	80,53	94,86	98,35	98,59	98,88	98,97	99,11	99,16
Run 3	8,64	29,56	55,55	82,08	94,12	98,55	98,74	98,94	98,97	99,12	99,13
Run 4	8,72	30,58	53,88	83,04	95,40	98,38	98,84	98,84	98,95	99,10	99,17
Run 5	8,71	30,69	55,99	86,23	95,97	98,41	98,67	98,83	99,01	99,12	99,13
Run 6	8,80	30,88	57,46	83,97	97,60	98,38	98,51	98,86	99,03	99,10	99,11
Run 7	8,56	30,48	49,36	85,25	95,61	98,25	98,69	98,91	99,06	99,13	99,16
Run 8	8,64	30,98	52,86	79,19	94,76	98,23	98,66	98,95	99,00	99,08	99,16
Run 9	8,71	29,90	57,88	81,10	95,20	98,31	98,76	98,78	99,04	98,99	99,22
Run 10	8,80	30,43	49,01	84,35	95,18	98,34	98,60	98,97	99,00	99,13	99,16
Average	**8,687**	**30,623**	**54,464**	**82,524**	**95,267**	**98,317**	**98,565**	**98,88**	**99,006**	**99,093**	**99,156**

WebSocket - CPU Load in Test Scenario 1

# clients	1	50	100	150	200	250	300	350	400	450	500
Run 1	8,01	30,09	46,03	70,83	80,90	89,69	95,84	98,37	98,57	98,02	98,81
Run 2	8,10	30,23	50,12	72,44	83,33	89,49	95,26	97,30	97,85	98,41	96,97
Run 3	7,94	29,94	45,85	70,89	80,68	90,00	95,12	97,47	98,92	97,33	98,45
Run 4	8,12	29,92	48,24	72,56	83,11	89,16	96,67	98,11	98,08	98,33	97,81
Run 5	8,03	30,73	47,99	70,24	80,99	91,40	96,16	98,18	97,49	98,69	97,72
Run 6	8,07	33,27	51,03	71,39	83,10	90,51	97,16	97,64	98,74	98,22	98,57
Run 7	8,04	30,03	53,82	70,26	81,13	89,46	96,22	97,57	98,79	98,75	98,90
Run 8	8,10	30,31	53,83	64,49	85,31	91,03	96,15	98,59	98,44	98,54	98,76
Run 9	8,07	30,17	50,66	66,99	81,36	89,76	95,75	97,85	98,74	98,68	98,62
Run 10	8,04	29,81	49,99	70,80	83,63	92,55	96,60	98,44	98,59	98,93	98,85
Average	**8,052**	**30,45**	**49,756**	**70,089**	**82,354**	**90,305**	**96,093**	**97,952**	**98,421**	**98,39**	**98,346**

HTTP Long Polling - Average Response Time in Test Scenario 1

# clients	1	50	100	150	200	250	300	350	400	450	500
Run 1	2,20	21,47	47,47	71,52	104,67	133,89	141,52	187,51	259,57	300,78	357,26
Run 2	2,16	21,10	46,68	71,47	101,19	131,05	135,93	189,05	259,32	303,22	390,97
Run 3	2,15	20,49	46,37	71,15	101,32	132,81	142,82	190,11	248,32	309,05	402,08
Run 4	2,19	20,16	46,89	70,01	99,85	136,70	146,21	194,01	266,06	300,87	365,01
Run 5	2,15	21,66	47,43	72,13	99,09	134,57	148,66	199,33	252,19	303,07	383,06
Run 6	2,19	21,65	47,35	70,28	100,15	127,86	140,74	187,31	283,20	306,13	352,76
Run 7	2,16	20,34	46,70	70,35	101,30	137,20	143,83	180,54	262,29	326,04	351,64
Run 8	2,21	19,97	46,54	70,49	98,98	139,12	141,29	196,23	257,18	297,92	381,64
Run 9	2,16	21,37	46,85	68,08	100,37	123,32	140,82	188,45	261,47	299,39	378,53
Run 10	2,17	21,11	46,78	73,10	99,06	131,20	147,39	177,88	244,94	299,01	359,56
Average	**2,174**	**20,932**	**46,906**	**70,858**	**100,598**	**132,772**	**142,921**	**189,042**	**259,454**	**304,548**	**372,251**

Server Sent Events - Average Response Time in Test Scenario 1

# clients	1	50	100	150	200	250	300	350	400	450	500
Run 1	2,33	3,19	4,18	5,45	9,30	392,48	394,46	1127,86	1003,25	2421,00	989,00
Run 2	2,37	3,30	4,17	5,70	10,09	587,77	492,25	1022,80	1878,40	1174,67	2020,00
Run 3	2,30	3,31	4,20	5,69	9,32	400,00	1419,14	1513,80	1564,00	1386,50	2540,50
Run 4	2,29	3,26	4,05	5,50	10,42	649,25	1099,40	936,75	1646,40	2201,25	2260,67
Run 5	2,31	3,31	4,04	6,00	11,77	564,42	1177,67	995,75	1006,67	1388,25	1723,25
Run 6	2,34	3,26	4,28	5,63	16,44	596,57	561,93	907,67	1269,75	1397,67	2582,00
Run 7	2,32	3,33	4,01	5,90	10,25	525,98	756,63	1936,00	1034,67	2920,67	1408,50
Run 8	2,33	3,26	3,71	5,31	9,46	460,51	732,90	1148,50	861,75	1225,67	2991,67
Run 9	2,38	3,20	4,09	5,26	10,03	784,52	830,00	720,25	2183,50	1154,25	985,50
Run 10	2,32	3,29	4,03	5,66	9,90	629,94	599,55	1501,00	1804,00	1634,00	1092,50
Average	**2,329**	**3,271**	**4,076**	**5,61**	**10,698**	**559,144**	**806,393**	**1181,038**	**1425,239**	**1690,393**	**1859,359**

WebSocket - Average Response Time in Test Scenario 1

# clients	1	50	100	150	200	250	300	350	400	450	500
Run 1	1,04	1,24	1,68	1,97	2,89	4,65	8,88	568,62	770,55	725,04	900,09
Run 2	1,04	1,24	1,50	1,90	3,27	5,06	11,64	686,52	715,13	891,17	767,34
Run 3	1,06	1,22	1,66	2,03	2,87	4,77	8,12	411,51	828,46	719,97	707,70
Run 4	1,02	1,16	1,52	1,94	2,98	4,71	15,90	507,47	824,01	405,13	826,00
Run 5	1,10	1,31	1,51	1,75	3,10	5,59	10,97	668,46	540,48	1111,92	862,21
Run 6	1,09	1,34	1,35	2,06	3,38	4,80	21,53	701,75	936,25	730,13	632,85
Run 7	1,06	1,40	1,42	1,94	2,93	4,55	9,71	378,00	1021,14	1087,83	886,50
Run 8	1,03	1,14	1,56	1,65	3,70	5,45	9,21	795,69	670,44	639,58	848,10
Run 9	1,06	1,24	1,42	1,69	2,92	4,55	9,26	646,17	857,54	791,46	740,71
Run 10	1,08	1,27	1,41	1,83	3,27	5,57	12,21	718,51	663,99	851,22	790,07
Average	**1,058**	**1,256**	**1,503**	**1,876**	**3,131**	**4,97**	**11,743**	**608,27**	**782,799**	**795,345**	**796,157**

Test Phase - Scenario 2

HTTP Long Polling - CPU Load in Test Scenario 2

# clients	1	50	100	150	200	250	300	350	400	450	500
Run 1	0,08	29,37	78,32	96,35	96,75	96,77	96,89	96,90	97,31	97,34	96,97
Run 2	0,09	29,58	78,23	96,60	96,74	96,89	96,88	96,96	97,19	97,29	97,12
Run 3	0,09	28,56	77,18	96,62	96,75	96,89	96,92	97,00	96,66	97,26	97,17
Run 4	0,09	28,99	77,64	96,39	96,71	96,13	96,87	96,98	96,99	97,31	97,18
Run 5	0,08	28,22	77,41	96,49	96,76	96,82	96,96	96,33	97,03	96,23	96,90
Run 6	0,09	28,83	77,58	96,53	96,57	96,81	96,87	96,93	97,29	97,16	97,09
Run 7	0,09	28,43	78,31	96,58	96,66	95,95	96,93	96,97	97,25	97,16	97,04
Run 8	0,09	28,49	77,87	96,49	96,81	96,93	96,58	96,94	97,20	97,27	97,27
Run 9	0,09	28,43	77,67	96,44	96,73	96,19	96,88	96,98	97,25	97,17	97,01
Run 10	0,08	28,95	78,46	96,44	96,81	96,82	96,10	96,93	97,28	97,17	97,05
Average	**0,087**	**28,785**	**77,867**	**96,493**	**96,729**	**96,62**	**96,788**	**96,892**	**97,145**	**97,136**	**97,08**

Server Sent Events - CPU Load in Test Scenario 2

# clients	1	50	100	150	200	250	300	350	400	450	500
Run 1	0,05	8,81	26,63	35,86	50,43	61,52	83,51	92,76	92,26	92,71	92,41
Run 2	0,05	8,83	26,04	40,75	50,71	64,42	84,21	92,64	92,72	92,04	91,92
Run 3	0,05	8,88	26,73	38,83	50,73	63,78	83,81	92,77	92,08	92,40	92,01
Run 4	0,06	8,82	26,51	36,02	50,26	64,45	83,86	92,76	92,27	92,29	92,45
Run 5	0,05	8,90	27,06	38,55	49,79	62,49	83,48	92,92	92,25	92,39	92,35
Run 6	0,06	8,80	26,41	42,35	49,29	63,39	83,27	92,73	92,69	92,53	92,91
Run 7	0,06	8,99	26,88	36,56	50,08	62,36	84,18	92,94	92,68	92,19	92,24
Run 8	0,04	8,86	26,45	35,03	49,79	64,18	84,76	92,70	92,30	92,25	92,17
Run 9	0,03	9,00	26,66	36,56	51,28	63,51	82,94	92,75	92,22	92,44	91,91
Run 10	0,06	8,96	26,00	40,44	50,25	62,68	86,31	92,74	92,12	92,44	92,17
Average	**0,051**	**8,885**	**26,537**	**38,095**	**50,261**	**63,278**	**84,033**	**92,771**	**92,359**	**92,368**	**92,254**

WebSocket - CPU Load in Test Scenario 2

# clients	1	50	100	150	200	250	300	350	400	450	500
Run 1	0,01	7,33	26,37	47,59	62,36	62,16	90,06	94,21	93,99	93,95	94,40
Run 2	0,01	7,33	26,23	25,60	63,33	77,39	90,79	94,06	94,18	94,53	93,84
Run 3	0,01	7,61	26,73	48,08	65,47	75,84	89,89	94,22	94,08	93,04	93,77
Run 4	0,01	7,26	26,57	47,82	56,99	64,38	91,89	94,17	93,92	93,84	94,31
Run 5	0,01	7,31	26,64	48,68	64,20	75,77	91,79	94,29	93,89	94,34	93,94
Run 6	0,01	7,47	26,59	48,28	40,08	64,55	88,95	94,35	94,28	94,49	93,74
Run 7	0,01	7,58	26,69	48,29	39,22	75,61	90,40	94,42	94,06	94,19	94,07
Run 8	0,01	7,54	26,68	47,98	62,13	76,12	90,99	94,21	94,12	94,04	93,79
Run 9	0,01	7,36	26,34	24,76	38,90	76,19	93,03	94,34	94,25	94,03	93,83
Run 10	0,01	7,19	27,34	48,12	63,01	63,64	91,66	94,44	94,09	94,25	94,32
Average	**0,01**	**7,398**	**26,618**	**43,52**	**55,569**	**71,165**	**90,945**	**94,271**	**94,086**	**94,07**	**94,001**

HTTP Long Polling - Average Response Time in Test Scenario 2

# clients	1	50	100	150	200	250	300	350	400	450	500
Run 1	5,00	10,38	28,32	136,08	179,02	233,03	298,59	371,20	431,14	562,67	732,19
Run 2	5,25	10,39	21,77	133,44	178,23	224,27	295,15	365,36	436,40	568,87	729,48
Run 3	5,13	10,09	22,47	132,62	178,62	231,64	300,78	367,17	467,06	556,17	719,08
Run 4	5,00	10,23	23,13	136,89	177,40	256,55	300,29	372,34	443,06	568,76	707,23
Run 5	5,00	10,05	22,24	133,63	182,11	231,58	299,26	404,52	448,97	658,70	725,45
Run 6	5,25	10,27	24,16	134,67	181,48	236,74	296,94	367,56	441,98	551,51	701,43
Run 7	5,00	10,10	25,00	133,43	181,38	266,19	294,38	367,34	438,21	565,30	702,98
Run 8	5,13	10,11	25,03	134,10	178,30	227,68	312,67	365,11	442,62	564,03	706,38
Run 9	5,50	10,07	23,93	133,75	181,50	256,47	294,89	365,66	437,04	582,09	719,73
Run 10	5,25	10,27	25,32	134,13	178,65	229,72	327,49	363,13	442,79	547,84	718,56
Average	**5,151**	**10,196**	**24,137**	**134,274**	**179,669**	**239,387**	**302,044**	**370,939**	**442,927**	**572,594**	**716,251**

Server Sent Events - Average Response Time in Test Scenario 2

# clients	1	50	100	150	200	250	300	350	400	450	500
Run 1	5,25	5,80	7,62	8,95	16,19	34,15	186,48	3004,89	3894,47	5331,73	5854,79
Run 2	5,38	5,77	7,81	9,26	13,84	36,93	204,74	2800,87	3977,96	5000,66	5858,34
Run 3	5,25	5,77	7,68	9,45	13,02	39,15	195,98	2874,00	4399,06	5204,05	6161,97
Run 4	5,25	5,74	7,74	9,61	11,17	44,36	166,61	2968,29	4375,28	5137,51	6022,98
Run 5	5,50	5,82	7,57	9,54	13,16	45,00	142,33	3032,16	4226,24	5440,15	5964,23
Run 6	5,38	5,77	7,82	9,51	14,58	35,29	181,05	2676,65	4187,49	5279,24	6260,23
Run 7	5,25	5,80	7,72	9,54	14,06	37,77	181,02	3112,02	3976,14	5230,53	5749,43
Run 8	5,38	5,77	7,56	9,74	14,28	37,80	185,38	2827,62	4298,76	5128,29	5820,35
Run 9	4,13	5,86	7,78	9,41	14,29	30,00	201,09	2802,05	4348,69	5339,55	5728,49
Run 10	4,38	5,80	7,55	9,54	12,98	34,09	195,44	3006,95	4138,52	5504,33	6052,50
Average	**5,115**	**5,79**	**7,685**	**9,455**	**13,757**	**37,454**	**184,012**	**2910,55**	**4182,261**	**5259,604**	**5947,331**

WebSocket - Average Response Time in Test Scenario 2

# clients	1	50	100	150	200	250	300	350	400	450	500
Run 1	1,75	3,22	5,22	7,47	9,46	12,91	127,41	4011,77	4496,76	4890,20	5648,91
Run 2	1,63	3,22	5,25	4,12	8,95	18,74	109,53	3758,67	4271,51	5875,74	6253,93
Run 3	1,75	3,34	5,27	7,12	9,07	16,42	74,76	3937,77	4847,48	5609,76	6303,20
Run 4	1,63	3,20	5,30	7,68	8,65	12,84	103,42	3966,14	4717,44	5036,62	5649,20
Run 5	1,88	3,23	5,34	7,24	9,26	17,09	88,62	3272,82	5433,95	5412,69	6146,90
Run 6	1,88	3,27	5,25	7,05	5,36	14,01	80,78	3534,59	5077,18	5490,90	6209,67
Run 7	1,63	3,28	5,25	7,27	5,16	12,56	91,42	3266,58	4914,74	4747,50	5279,45
Run 8	2,00	3,32	5,27	7,23	9,25	14,79	113,96	3996,53	4491,31	5896,76	5822,92
Run 9	1,75	3,26	5,24	3,99	5,20	14,93	132,67	3141,57	5182,45	6161,56	6283,20
Run 10	1,75	3,20	5,39	7,39	9,07	11,37	103,31	3643,02	4710,91	5316,17	6013,30
Average	**1,765**	**3,254**	**5,278**	**6,656**	**7,943**	**14,566**	**102,588**	**3652,946**	**4814,373**	**5443,79**	**5961,068**

HTTP Long Polling - Median Response Time in Test Scenario 2

# clients	1	50	100	150	200	250	300	350	400	450	500
Run 1	3,00	10,00	19,00	133,50	175,00	232,00	301,50	375,00	433,00	567,00	733,00
Run 2	4,00	10,00	18,00	129,00	174,50	218,00	293,75	369,50	436,75	575,00	727,00
Run 3	3,50	10,00	17,50	130,00	178,00	229,00	301,00	370,00	472,00	558,00	719,00
Run 4	3,00	9,00	17,00	135,50	173,00	255,00	302,00	372,00	436,00	573,00	712,50
Run 5	3,00	10,00	18,00	131,00	176,00	230,00	295,00	397,00	446,00	666,25	726,50
Run 6	3,50	10,00	18,00	130,00	177,00	236,00	295,00	367,50	440,50	552,00	705,75
Run 7	3,00	10,00	18,00	128,00	176,00	264,00	296,00	362,50	433,00	569,00	707,50
Run 8	4,00	9,00	18,00	131,50	176,00	226,00	304,50	356,00	443,00	566,50	713,00
Run 9	4,00	9,00	19,00	129,00	178,00	258,25	294,00	359,50	432,00	585,00	720,50
Run 10	4,00	10,00	18,00	130,00	176,00	225,00	328,00	360,00	436,00	549,00	722,00
Average	**3,5**	**9,7**	**18,05**	**130,75**	**175,95**	**237,325**	**301,075**	**368,9**	**440,825**	**576,075**	**718,675**

Server Sent Events - Median Response Time in Test Scenario 2

# clients	1	50	100	150	200	250	300	350	400	450	500
Run 1	3,00	5,00	6,00	7,00	9,00	13,00	156,00	2671,00	4090,50	5031,00	5458,50
Run 2	3,00	5,00	6,00	8,00	9,00	14,00	194,00	2601,50	3729,00	4446,00	5115,00
Run 3	3,50	5,00	6,50	7,50	9,00	18,00	184,00	2666,50	3819,50	4810,50	5405,00
Run 4	3,00	5,00	6,00	7,00	8,00	16,00	162,00	2805,00	4041,50	4734,00	5542,00
Run 5	3,00	5,00	6,00	7,50	9,00	14,00	124,00	2670,00	3904,50	4757,00	5681,00
Run 6	3,00	5,00	6,00	8,00	9,00	13,00	169,00	2465,00	3872,00	4870,50	5779,00
Run 7	3,00	5,00	6,25	8,00	9,00	13,00	163,00	2893,50	3740,00	4897,00	5009,00
Run 8	3,00	5,00	6,00	7,00	9,00	14,00	176,00	2656,00	4053,00	4607,00	5216,00
Run 9	3,00	5,00	6,00	8,00	9,00	13,00	191,00	2592,50	3808,50	4738,00	4936,00
Run 10	3,50	5,00	6,00	8,00	9,00	14,00	184,50	2689,50	3800,00	4907,00	4892,00
Average	**3,1**	**5**	**6,075**	**7,6**	**8,9**	**14,2**	**170,35**	**2671,05**	**3885,85**	**4779,8**	**5303,35**

WebSocket - Median Response Time in Test Scenario 2

# clients	1	50	100	150	200	250	300	350	400	450	500
Run 1	1,00	3,00	5,00	7,00	9,00	6,00	128,00	2439,50	3540,00	3800,00	4376,00
Run 2	1,00	3,00	5,00	3,00	9,00	11,00	118,53	2733,00	3437,00	4431,00	4781,00
Run 3	1,00	3,00	5,00	7,00	9,00	11,00	73,00	2288,50	3595,00	3947,50	4840,50
Run 4	1,00	3,00	5,00	7,00	8,00	5,00	104,00	2503,00	3268,00	4084,50	4154,00
Run 5	1,50	3,00	5,00	7,00	9,00	12,00	91,00	2322,50	3631,00	4228,00	3969,00
Run 6	1,50	3,00	5,00	7,00	4,00	6,00	79,00	2174,00	3263,50	3662,00	4263,00
Run 7	1,00	3,00	5,00	7,00	4,00	11,00	93,00	2235,50	3912,00	3512,00	4168,00
Run 8	2,00	3,00	5,00	7,00	9,00	11,00	120,00	2376,00	3571,00	3741,50	4584,00
Run 9	1,00	3,00	5,00	3,00	4,00	11,00	139,00	2158,50	3074,00	4426,00	4836,00
Run 10	1,00	3,00	5,00	7,00	9,00	5,00	107,00	2021,00	3429,00	4267,00	4577,00
Average	**1,2**	**3**	**5**	**6,2**	**7,4**	**8,9**	**105,253**	**2325,15**	**3472,05**	**4009,95**	**4454,85**

Memory Footprint after Test - Scenario 1

HTTP Long Polling - Memory Footprint after Test Scenario 1

# clients	1	50	100	150	200	250	300	350	400	450	500
Run 1	43,41	63,36	65,68	67,92	70,83	73,17	78,64	89,17	89,23	88,75	96,98
Run 2	43,05	63,66	65,37	67,36	70,72	73,78	78,08	88,44	89,21	97,54	98,05
Run 3	43,34	62,57	66,03	67,51	70,50	74,37	77,78	89,17	89,48	88,27	98,95
Run 4	43,53	62,70	66,00	68,20	69,93	74,93	78,45	89,58	89,26	96,72	97,40
Run 5	43,82	64,52	65,70	67,73	71,15	73,61	76,10	89,07	98,31	88,02	97,24
Run 6	43,12	64,03	65,87	67,64	71,08	75,77	77,45	89,22	88,85	89,02	96,96
Run 7	43,43	62,81	65,92	67,73	70,07	73,21	78,34	87,95	88,23	90,10	98,44
Run 8	43,68	63,19	66,08	67,49	70,44	74,71	77,60	89,24	89,28	89,62	98,38
Run 9	43,80	63,86	66,37	67,07	71,15	73,91	77,18	88,62	96,76	98,04	98,40
Run 10	43,41	63,34	65,59	68,66	70,22	73,73	76,79	88,74	88,15	89,35	97,90
Average	**43,459**	**63,404**	**65,861**	**67,731**	**70,609**	**74,119**	**77,641**	**88,92**	**90,676**	**91,543**	**97,87**

Server Sent Events - Memory Footprint after Test Scenario 1

# clients	1	50	100	150	200	250	300	350	400	450	500
Run 1	41,02	40,75	41,56	42,45	59,08	361,25	697,65	893,30	1056,93	1094,48	1272,93
Run 2	40,54	41,18	41,54	42,43	59,09	357,21	753,91	871,86	988,80	1187,88	1331,42
Run 3	40,69	40,90	41,51	42,12	59,10	555,30	742,80	906,60	1074,58	1241,28	1208,90
Run 4	41,23	40,69	41,99	42,06	58,85	229,34	751,12	930,91	967,50	1179,23	1266,82
Run 5	41,54	41,01	41,83	42,44	59,20	162,64	707,84	868,05	1062,02	1150,16	1310,55
Run 6	40,77	41,27	41,58	42,11	59,36	312,22	690,46	863,07	1113,82	1180,38	1282,28
Run 7	40,42	41,07	41,65	42,21	58,99	259,49	759,35	811,11	1022,30	1106,56	1313,60
Run 8	41,16	40,79	41,67	42,35	58,75	98,32	738,63	857,99	1087,98	1129,83	1296,55
Run 9	41,23	40,84	41,50	41,98	59,30	297,66	679,59	822,48	975,38	1084,34	1296,39
Run 10	41,74	40,99	41,42	42,00	59,08	286,76	711,05	857,06	1048,90	1141,94	1291,83
Average	**41,034**	**40,949**	**41,625**	**42,215**	**59,08**	**292,019**	**723,24**	**868,243**	**1039,821**	**1149,608**	**1287,127**

WebSocket - Memory Footprint after Test Scenario 1

# clients	1	50	100	150	200	250	300	350	400	450	500
Run 1	22,68	29,52	39,78	42,04	63,59	65,08	70,19	1029,58	1316,49	1248,57	1397,89
Run 2	22,59	29,59	40,02	41,81	62,50	64,99	69,01	940,06	1260,37	1616,52	976,69
Run 3	22,76	29,50	40,11	41,62	62,22	64,90	68,10	696,34	1246,16	1230,47	1665,54
Run 4	22,78	29,50	39,89	42,05	62,31	64,82	74,66	726,87	1129,14	1167,72	1400,90
Run 5	22,64	29,57	40,13	41,46	62,58	65,02	69,92	948,68	1075,36	1473,15	978,56
Run 6	22,68	29,40	39,90	41,47	62,66	65,10	72,35	799,30	1283,27	1387,47	1598,40
Run 7	22,71	29,34	39,99	41,42	62,59	64,49	71,05	222,08	1307,06	1451,38	1801,90
Run 8	22,83	29,34	39,94	43,26	63,36	64,32	70,05	976,62	1339,91	1337,47	1820,96
Run 9	22,84	29,58	39,87	42,90	63,27	65,13	68,47	819,69	1317,93	1193,72	1717,20
Run 10	22,84	29,52	39,85	41,61	63,05	64,94	69,94	931,28	1354,81	1587,94	1754,12
Average	**22,735**	**29,486**	**39,948**	**41,964**	**62,813**	**64,879**	**70,374**	**809,05**	**1263,05**	**1369,441**	**1511,216**

Memory Footprint after Test - Scenario 2

HTTP Long Polling - Memory Footprint after Test Scenario 2

# clients	1	50	100	150	200	250	300	350	400	450	500
Run 1	36,27	67,10	67,69	68,67	71,25	71,66	74,34	78,12	80,07	81,71	82,88
Run 2	36,40	67,00	68,32	68,06	68,49	71,56	73,22	77,94	77,74	81,30	82,88
Run 3	36,04	66,56	68,32	67,48	70,02	70,98	74,61	78,31	79,46	78,40	82,68
Run 4	36,34	66,44	68,96	67,62	69,64	70,14	74,38	77,65	78,44	80,04	83,96
Run 5	36,19	66,48	68,15	67,32	68,70	72,30	75,26	76,44	82,00	80,32	85,18
Run 6	36,23	67,05	67,01	67,60	69,68	72,98	73,39	77,98	80,15	81,34	82,43
Run 7	36,04	66,40	68,14	67,30	69,09	71,04	74,24	76,46	77,80	80,72	83,50
Run 8	36,16	66,40	68,13	67,80	69,33	71,62	74,18	78,62	78,96	81,41	89,95
Run 9	36,20	66,78	68,46	68,46	69,67	71,59	74,66	77,54	80,00	80,15	83,25
Run 10	36,33	66,55	68,06	68,35	68,85	73,46	74,76	77,31	78,46	82,55	88,25
Average	**36,22**	**66,676**	**68,124**	**67,866**	**69,472**	**71,733**	**74,304**	**77,637**	**79,308**	**80,794**	**84,496**

Server Sent Events - Memory Footprint after Test Scenario 2

# clients	1	50	100	150	200	250	300	350	400	450	500
Run 1	35,73	51,65	68,10	67,23	70,43	71,58	77,79	97,82	99,68	126,90	125,63
Run 2	35,84	53,24	68,08	68,60	69,30	71,64	77,74	95,78	103,48	111,45	130,34
Run 3	35,71	51,16	67,60	67,62	69,01	72,45	77,14	97,76	104,28	109,30	133,86
Run 4	35,66	51,06	68,28	67,74	68,67	72,41	77,91	96,00	102,48	116,38	131,14
Run 5	35,85	53,66	68,02	68,63	69,28	71,89	76,95	95,50	100,73	112,83	119,54
Run 6	35,93	53,22	67,98	67,39	69,38	71,87	77,27	96,53	101,36	114,02	124,94
Run 7	35,99	52,79	67,88	67,57	69,15	71,88	77,67	97,30	100,42	108,30	122,42
Run 8	35,92	48,86	67,75	67,72	69,38	71,64	77,49	97,31	102,62	113,14	120,10
Run 9	35,96	52,68	67,17	67,65	69,40	71,19	78,10	98,87	101,48	111,55	126,96
Run 10	36,07	49,98	68,16	67,78	69,16	72,64	77,73	97,58	103,18	118,65	134,90
Average	**35,866**	**51,83**	**67,902**	**67,793**	**69,316**	**71,919**	**77,579**	**97,045**	**101,971**	**114,252**	**126,983**

WebSocket - Memory Footprint after Test Scenario 2

# clients	1	50	100	150	200	250	300	350	400	450	500
Run 1	18,66	25,58	34,94	45,56	50,62	63,90	69,60	286,43	283,70	358,87	333,42
Run 2	18,56	25,68	33,95	45,32	50,34	64,55	69,38	277,82	307,42	458,03	469,24
Run 3	18,69	25,66	34,03	44,78	50,15	64,60	68,18	292,95	269,18	385,39	470,64
Run 4	18,67	25,54	34,24	45,18	49,90	64,14	71,09	301,92	273,38	337,76	332,54
Run 5	18,79	25,27	34,64	44,96	50,24	64,36	69,81	193,61	390,38	420,72	439,95
Run 6	18,61	25,21	34,13	45,11	49,94	64,44	68,44	258,71	383,47	442,60	458,38
Run 7	18,82	25,40	34,67	45,22	50,31	64,02	70,14	263,57	320,68	330,32	348,32
Run 8	18,65	25,37	34,84	45,19	50,15	64,22	70,52	295,07	289,42	422,57	304,79
Run 9	18,83	25,48	34,45	45,01	50,09	64,06	70,80	259,86	370,68	363,57	471,20
Run 10	18,83	25,48	34,34	45,12	49,96	64,44	68,88	288,08	254,62	420,92	402,34
Average	**18,711**	**25,467**	**34,423**	**45,145**	**50,17**	**64,273**	**69,684**	**271,802**	**314,293**	**394,075**	**403,082**