

Real-Time Web with WebSocket

A Comparison of Real-Time Technologies for the Web Using Node.js as Platform

Contents

| | |
|---|------------------|
| Contents | 2 |
| List of Figures | 6 |
| List of Tables | 7 |
| List of Examples | 7 |
| Abstract | 8 |
| Preface | 9 |
| Acknowledgements | 10 |
| <u>Chapter 1: Introduction</u> | <u>11</u> |
| 1.1 Motivation | 11 |
| 1.2 Research Questions | 12 |
| 1.3 Outline | 12 |
| <u>Chapter 2: Background</u> | <u>14</u> |
| 2.1 Related Academic Work | 14 |
| 2.1.1 Johannessen's Thesis' Relation To This Thesis | 15 |
| 2.2 TCP | 15 |
| 2.3 The Web and HTTP | 16 |
| 2.3.1 The World Wide Web | 16 |
| 2.3.2 HTTP | 16 |
| 2.3.3 HTTP Methods | 16 |
| 2.3.4 HTTP Header Fields and State | 17 |
| 2.4 Modern Web | 18 |
| 2.4.1 Ajax | 18 |
| 2.5 Real-Time HTTP | 19 |
| 2.5.1 HTTP Polling | 19 |
| 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 WebSockets vs. HTTP | 29 |

| | |
|--|-----------|
| 2.8 Web Platforms & Node.js | 31 |
| 2.8.1 Web Development | 31 |
| 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 | 35 |
| 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 | 45 |
| 3.5.2 Test Scenario 2 | 47 |
| 3.6 Development | 49 |
| 3.6.1 Common Between Scenarios | 49 |
| 3.6.2 Software Versions | 51 |
| 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 | 55 |
| 3.7.2 Network Use | 55 |
| 3.7.3 Quality and Correctness of the Code | 55 |
| 3.7.4 Only One Software Platform | 55 |
| 3.7.5 Node.js | 55 |
| Chapter 4: Test Results | 57 |
| 4.1 Idle Client Phase | 57 |

| | |
|---|-----------|
| 4.1.1 CPU Load | 58 |
| 4.2.2 Memory Footprint | 59 |
| 4.2.3 Response Time | 60 |
| 4.3 Test Phase - Scenario 1 | 61 |
| 4.3.1 CPU Load During Broadcast | 61 |
| 4.3.2 Response Time During Broadcast | 62 |
| 4.4 Test Phase - Scenario 2 | 63 |
| 4.4.1 CPU Load During Chat | 63 |
| 4.4.2 Response Time During Chat | 64 |
| 4.5 Memory Footprint After Tests | 66 |
| 4.6 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 | 74 |
| 5.4.1 Issue With the WebSocket Implementation | 74 |
| 5.4.2 Node.js | 74 |
| 5.4.3 Node.js' HTTP Is More Tested and Stable | 75 |
| 5.4.4 Errors with the Test Implementation | 75 |
| 5.5 Implementation | 75 |
| 5.5.1 Test Scenario 1 | 76 |
| 5.5.2 Test Scenario 2 | 77 |
| Chapter 6: Ending | 78 |
| 6.1 Thesis Conclusion | 78 |
| 6.2 Further Work (in discussion?) | 79 |
| Bibliography | 80 |
| Appendix | 83 |
| List of Acronyms | 83 |
| Code | 83 |
| Software Versions | 84 |

| | |
|--|----|
| How to Run the Tests | 84 |
| 3.9.1 Scenario 1 | 84 |
| 3.9.2 Scenario 2 | 84 |
| Full Test Results | 85 |
| Idle Clients Phase | 85 |
| Test Phase - Scenario 1 | 89 |
| Test Phase - Scenario 2 | 91 |
| Memory Footprint After Test - Scenario 1 | 94 |
| Memory Footprint After Test - Scenario 2 | 95 |

List of Figures

| | |
|--|----|
| Figure 1: HTTP GET request to www.uio.no. | 17 |
| 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: Example of client side outdated data with Long Polling. | 22 |
| Figure 6: The EventSource API. | 23 |
| Figure 7: The WebSocket API. | 26 |
| Figure 8: The WebSocket frame. | 28 |
| 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. | 52 |
| 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. | 54 |
| 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: The CPU load for all three transports during the idle phase. | 58 |
| Figure 25: The memory footprint for all three transports during the idle phase. | 59 |
| Figure 26: The response times for all three transports during the idle phase. | 60 |
| Figure 27: The CPU load during the first test scenario's test phase. | 61 |
| Figure 28: The response times during the first test scenario's test phase. | 62 |
| Figure 29: The CPU load during the seconds test scenario's test phase. | 63 |
| Figure 30: The response times during the second test scenario's test phase. | 64 |
| Figure 31: The response times during the second test scenario's test phase. | 65 |
| Figure 32: The memory footprint right after the first test scenario's test phase. | 66 |
| Figure 33: The memory footprint right after the second test scenario's test phase. | 67 |
| Figure 34: Scenario 1 load testing response time. | 70 |
| Figure 35: Scenario 2 load testing response time. | 70 |
| Figure 36: Scenario 1 stress testing response time. | 72 |
| Figure 37: Scenario 2 stress testing response time. | 72 |
| Figure 38: The three different servers in the first test scenario | 76 |
| Figure 39: The three different servers in the second test scenario. | 77 |

List of Tables

| | |
|--|----|
| Table 1: Ranking of the three transports in both test scenarios. | 67 |
| Table 2: The software versions used in this thesis | 84 |

List of Examples

| | |
|---|----|
| Example 1: Socket programming example. | 16 |
| Example 2: A GET request example. | 17 |
| Example 3: SSE connection example. | 23 |
| Example 4: Server Sent Events example. | 24 |
| Example 5: How to open a WebSocket connection. | 26 |
| Example 6: WebSocket opening handshake example. | 27 |
| Example 7: WebSocket connection with subprotocols. | 28 |
| Example 8: WebSocket extension. | 29 |
| Example 9: Stock price example JSON object. | 29 |
| Example 10: HTTP response with headers. | 30 |
| Example 11: HTTP request headers. | 30 |
| Example 12: Server Sent Events message example. | 30 |
| Example 13: Blocking code. | 32 |
| Example 14: Non-blocking asynchronous code. | 32 |
| Example 15: How to install a package to your Node.js project. | 33 |
| Example 16: How to allow 1024 user processes in OS X. | 43 |
| Example 17: Server Sent Events Endpoint from the test code. | 50 |

Abstract

Text

Preface

Text

Acknowledgements

Text

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 transformation to the rich and dynamic web of today, is quite amazing. Today, there are only small differences between applications running locally on your device and applications running in a web browser - web applications. Part of what makes this possible is the fact that the web has become capable of real-time updates. The *real-time web* makes it possible for a website to dynamically update itself when new content is available.

To better understand what is meant by the term real-time in this thesis, consider the following two examples.

1. A stock price application where several clients are connected to server. This stock price server subscribes to a broker backend for updates on stock prices. When the stock price server receives price updates from the backend, it broadcasts this to all clients.
2. A chat room application where several clients are connected 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.

Real-time means that the data is handled and distributed *immediately* after it is received. This is achieved by the concept of a persistent connection and a socket that one can push data to at any time.

The first example requires unidirectional messaging, with stock price messages only going server-to-client. The second example is bidirectional, requiring both server-to-client and client-to-server messaging. These two examples show the types of real-time applications this thesis focuses on.

The web protocol HTTP was not designed with real-time in mind, but with the new protocol *WebSocket* and the new HTTP based technology *Server Sent Events*, a real-time web is possible. There are also several clever ways to use traditional HTTP in order to achieve *near real-time*. Near real-time in this thesis means a solution that makes the user believe the application is real-time, even though it in reality is not. The two most notable HTTP based near real-time technologies are *HTTP Long Polling* and *HTTP Streaming* (both will be explained in detailed in chapter 2). WebSocket, Server Sent Events and Long Polling are the (near) real-time technologies this thesis focuses on. Server Sent Events and Long Polling are both unidirectional, server-to-client, while WebSocket is bidirectional, server-to-client and client-to-server. I will go through each of these in depth in chapter 2.

For simplicity's sake, throughout the thesis, I commonly refer to these (near) real-time delivering technologies as *transports*.

Note: In addition to the two real-time application types above, there is peer-to-peer. With WebRTC[1] peer-to-peer is possible on the web as well. However, as of this writing, WebRTC was not fully standardized, so I decided not to include WebRTC in this thesis.

1.2 Research Questions

The goal of this thesis is to compare WebSocket to the two HTTP real-time transports Long Polling and Server Sent Events. I will compare them based on two grounds, performance and programmer friendliness.

Performance is typically measured from two angles. First, from the server's view we have CPU load and memory consumption. Second is the user's perspective with response time. Further details on performance can be found in section 2.9.

Then, I will discuss differences between the three transports from a programmer's perspective. Are there any conceptual differences between the three transports? Which one is the easiest to use as a developer? Which one is most flexible and provide most functionality?

I have formed the following research questions for this thesis:

1. How does WebSocket perform compared to Long Polling and Server Sent Events in an unidirectional messaging setting with high levels of load?
2. How does WebSocket perform compared to Long Polling and Server Sent Events in an bidirectional messaging setting with high levels of load?
3. Does WebSocket provide any advantages over Long Polling and Server Sent Events from a programmer's perspective?
4. For what types of real-time web applications does WebSocket provide a benefit over Long Polling and Server Sent Events?

The last question is the main research question and the first three are meant to validate and substantiate it.

The main research question demonstrate my expectation that WebSocket is going to perform better than the other two transports. That expectation derives from the results found in Kristian Johannessen's 2014 master's thesis[2]. In fact, many of the choices made in this thesis is based on the work by and suggestions from Johannessen. His thesis and other related work is presented at the start of chapter 2.

1.3 Outline

Chapter 2 - Background

Chapter 2 presents all background material needed to understand the methodology and research part of this thesis. That means all three transports, the chosen software platform and a quick note on performance testing. Previous and other relevant academic work is also presented here.

Chapter 3 - Methodology

Based on the background material, I have developed two distinct real-time scenarios to compare the three different transports. This chapter presents these two scenarios and discusses parameters and choices. How the test results are retrieved is also discussed.

Chapter 4 - Results

In this chapter, the results will be presented as they were found when testing. The results are discussed and analyzed in chapter 5.

Chapter 5 - Discussion

This chapter includes detailed analysis and discussion of the results from the previous chapter. Any abnormalities will be explained with possible explanations.

Chapter 6 - Ending

The thesis ending consist of thesis conclusion that directly answers the research questions and a section of further work.

The thesis is concluded by a reference list and an appendix.

Chapter 2: Background

After deciding the research questions and a basic thesis premise, I needed to read a lot of material to get a great understanding of all the technologies I had to use. Most importantly, I needed to get a very solid understanding of how the Web works. Not only in terms of how webpages are built, with HTML documents, styled by CSS stylesheets and set in action by JavaScript code. But also how web pages are delivered from a web server to a web browser. This is done by an application layer protocol called HTTP.

With a solid understanding of the Web basics, I could start to learn about the modern Web with real-time capabilities. The real-time Web is either powered by clever techniques using HTTP or more modern ways with Server Sent Events or WebSocket.

Also key to this thesis was to understand how Node.js, my server platform of choice, works under the hood. Lastly, I needed some basic knowledge of what performance testing is, and what to look for in the test data.

This chapter will present all these technologies and techniques in detail, making the methodology, results and discussion chapters understandable. First in this chapter is a section on related academic work.

2.1 Related Academic Work

In the 2007 paper[3] Bozdag, Mesbah and Deursen compares 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 authors conclude with praise for the push approach if high data coherence and network performance is desired. But they do point out some problems with regard to scalability. The push approach they recommend is HTTP streaming. Server Sent Events and WebSocket are other push approaches, but are not mentioned, as they did not exist at the time when the article was written.

The bachelor's thesis by Jõhvik[4], seems to be heavily inspired and motivated by the work of Bozdag, Mesbah and Deursen. Jõhvik determines that the pull based HTTP Long Polling technique is better than HTTP streaming, conflicting with the views of the previous article. WebSocket did exist at the time of the article's writing, but Jõhvik decided not to include it because of browser incompatibility.

Then, there is the master's thesis by Kristian Johannessen[2]. In the first part of his thesis he compares different real-time frameworks for the web. Some frameworks such as SignalR and Lightstreamer, support several real-time transports, like Server Sent Events and WebSocket. Some even provide fallback solutions to support the largest possible set of clients. Based on performance and programmer friendliness (maturity, browser support, WebSocket support and presentation), he recommends SignalR as the best real-time framework, closely followed by Socket.IO.

The second part of his thesis compares WebSocket to traditional HTTP techniques for real-time behavior. He concludes by saying that “...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.

Because of similarities to Johannessen’s thesis, I had contact with him when writing. He said he met challenges when using full fledged web browsers as clients and consequently advised me to use smaller, lightweight command line clients for my tests. Because he chosen to use web browser clients, he could not focus on scalability and see what happens when the transports behave when the servers are pushed to resource limits.

2.1.1 Johannessen’s Thesis’ Relation To This Thesis

Based on Johannessen’s recommendations and inspiration from his thesis, I came up with a similar, yet distinctly different thesis. I wanted to focus solely on the transports - not the frameworks. I also wanted to use smaller command line clients that lets me spawn them in the number of hundreds and see how the server behaves when pushed to the limit.

The rest of this chapter includes all the technologies I had to learn before creating my test scenarios. That includes the protocols HTTP and WebSocket, as well as HTTP techniques like HTTP Long Polling for real-time behavior. Node.js, the chosen software platform is also presented. Lastly, a section on performance testing.

2.2 TCP

Before going into the Web and HTTP, TCP needs an introduction. TCP is short for Transmission Control Protocol and is one of the most important protocols on the internet, as it together with IP forms the foundation of the Internet Protocol Suite[5]. The protocol is not directly a part of this thesis, but knowing what it is and its characteristics is essential when reading further on. Both HTTP and WebSocket, which will be presented later in this chapter, runs as application layer protocols on top of TCP.

The most important aspects of TCP:

- TCP is a *transport layer protocol*. This 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 every packet will eventually arrive at the receivers end. This is ensured by having the receiver acknowledge 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. The socket is an data structure abstraction that can be both written and read from and treated like a file. This is very powerful and in addition to being full-duplex, TCP is perfect for real-time applications. The example below show pseudocode (and a simplification) that demonstrates how easy it is 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 example.

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 stylesheets 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 describe styling attributes of HTML components.
- JavaScript - The web's programming language.

The web still works around 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 a great example. Even though Google Maps seems to be completely different from simple websites such as blogs or newspapers, they are both powered by the same technologies underneath. Today it is very likely that several of the applications on your own smartphone are powered by HTML, CSS and JavaScript as a web application running locally on your device. This shows how far these web technologies has come.

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

HTML5[6] 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 totally 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 powering the Web. It is the protocol that is used to send web sites and all its components to a web browser. Being request-response oriented, means that all server-to-client messaging must be a response to a certain request. HTTP has many types of request types, 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. The server parses this request and responds with the requested information, or does the requested work. Each response is marked with a status code to indicate whether the request was successful or not.

GET was the first HTTP method[7] and is the one you send to a web server to request a certain file or document. The following example 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 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. This means that the server does not store any information regarding the clients. This is great thing in terms of server resources, but it can also be a problem, as we shall see.

Header fields are an important part of HTTP. They are meta data that are added into the HTTP requests and responses. The «Host» line in example above is the Host header field. Headers are there for the server and client to tell the other part a bit about themselves. As an example, it is useful for the server to know what kind of language the client wants the requested document in.

The example below, captured by HTTPScoop[8], shows the entire GET request a web browser sends when going to www.uio.no. The web browser adds several header fields:

| Request Headers | |
|-----------------|--|
| Name | Value |
| Accept | text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8 |
| Accept-Language | nb-no |
| Connection | keep-alive |
| User-Agent | Mozilla/5.0 (Macintosh; Intel Mac OS X 10_9_1) AppleWebKit/53... |
| Accept-Encoding | gzip, deflate |
| Cookie | _utma=161080505.1314720432.1383523269.1392032968.1... |
| Host | www.uio.no |

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

The browser adds a significant number of header fields. For example, the User-Agent field tells the server what kind of computer, OS 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 the request with a HTTP response. The response embody 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 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. Below is an example of a server response with its headers:

| Response Headers | |
|-------------------|-------------------------------|
| Name | Value |
| Content-Encoding | gzip |
| Server | Apache/2.2.25 (Unix) |
| Vary | Cookie |
| Cache-Control | max-age=300 |
| Content-Language | no |
| Date | Wed, 12 Feb 2014 11:53:50 GMT |
| Transfer-Encoding | chunked |
| Connection | keep-alive |
| X-Cache | MISS |
| Content-Type | text/html; charset=utf-8 |
| X-Varnish | 1257927478 |
| X-Cacheable | NO:Not Cacheable |
| Age | 0 |
| Via | 1.1 varnish |

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

2.4 Modern Web

As mentioned, HTTP was designed to serve static hyperlinked documents, but today you rarely visit a site that is static and pure HTML. Almost every site you visit is highly interactive with lots of JavaScript code running in the background. This development started in the late 1990s with Microsoft Outlook, but skyrocketed after 2004 with Google Gmail and Google Maps. These types of websites started to behave more like platform native applications and was a clear departure from the hyperlinked documents that the web originally consisted of. Terms like *Web Applications* and *Single-Page Apps* came forth, and with websites going more complex and JavaScript heavy, having a fast web browser was a clear advantage. This lead to a great race between Microsoft, Google, Mozilla and Apple to build the greatest JavaScript engines.

Essential to this development was how HTTP was used in order to achieve dynamic updates - Ajax.

2.4.1 Ajax

Ajax (Asynchronous JavaScript and XML) as a term was introduced by Jesse James Garret in 2004[9]. In it he states that Ajax is “several technologies, each flourishing in its own right, coming together in powerful new ways”. At the center of Ajax is the *XMLHttpRequest* JavaScript API[10]. 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 happens 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. Today JSON[11] is a widely used format for representing hierarchical key-value data, with less overhead compared to XML. A great example of an Ajax powered web application would be Google Maps.

When you pan around the map, the JavaScript running in your browser initiates Ajax GET requests to the server, requesting data of the area you are now looking at. When new images and map data has arrived, JavaScript running in your browser updates the DOM.

Ajax is at the heart of web applications, and it brought interactivity to an otherwise static web. Together with some clever techniques, Ajax can make the Web real-time.

2.5 Real-Time HTTP

For many applications, pushing data between server and client is essential. Let us say you have 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 be nice to push the update immediately to connected clients. Achieving this kind of push behavior is quite trivial for platform native applications since 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 and there is no native way for the server to push messages to the client. All server-sent messages must be a response to a client-sent request. Then, how can web applications like Twitter and Facebook seem as though they have real time server push capability? The chat on Facebook seems to push messages immediately. Following are a set of techniques, that accomplishes server push and near real-time using HTTP.

2.5.1 HTTP Polling

The first solution is having the client side JavaScript periodically polling 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 easy 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 update. This 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. This can vary from some seconds to even minutes if the message is long. Trying to find the perfect update request rate is really difficult and varies greatly from application to application. The worst case scenario is that you end up sending a lot of requests that return an empty response. This is undeniably a bad thing, as it congests the network with unnecessary traffic.

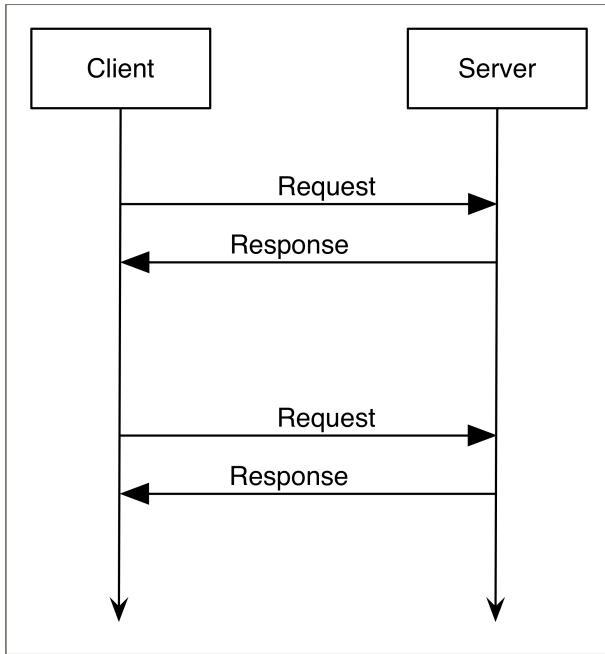


Figure 3: HTTP Polling example.

2.5.2 Comet

When you need real time server-push in your web app, HTTP Polling seems as a poor choice. Comet is an umbrella term for a set of programming models that achieve server push behavior only using existing HTTP technologies. The term Comet was first introduced by Alex Russell in a blogpost[12] he wrote in 2006. The two most used Comet techniques are HTTP Long Polling and HTTP Streaming.

HTTP Long Polling

HTTP Long Polling is essentially the same as regular HTTP Polling except that the server delays the response until either new data is ready to be sent or a timer runs out. Immediately after the response is received, the client sends a new request and waits for new updates. As default the timer is 45 seconds[3]. Long Polling gives the user the impression of having data pushed from the server, even though it in theory is not.

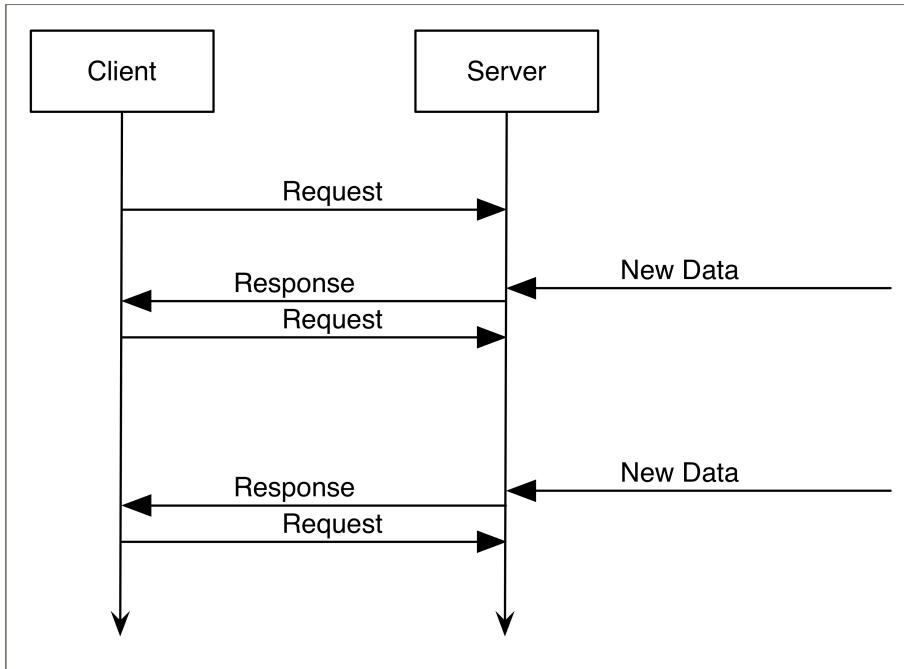


Figure 4: HTTP Long Polling Example.

HTTP Streaming

HTTP Streaming, also known as “the forever-frame”, is another practice that emulates server push. Chunked Encoding is a part of the HTTP/1.1 specification that lets the server start pushing chunked data to the client before the response size is known. A forever-frame is an HTML iframe that keeps receiving script tags as these chunks. These script tags are immediately executed on the client 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 seems to give web apps real-time pushing of data, why do we need Server Sent Events and WebSocket? For HTTP Streaming, the biggest problem is the fact that it is very hard to debug and error check. With the server pushing scripts that are immediately executed on the client, debugging can be very tricky. Security is also a concern, as the scripts are immediately executed.

For HTTP Long Polling, consider our stock price web app from earlier, with clients now using Long Polling. In 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. 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. Of course you can delegate the responsibility over to the client. A header field can indicate what the last received update was, so the server know what updates to send. But all this would have been a non-issue if there was a persistent stateful connection.

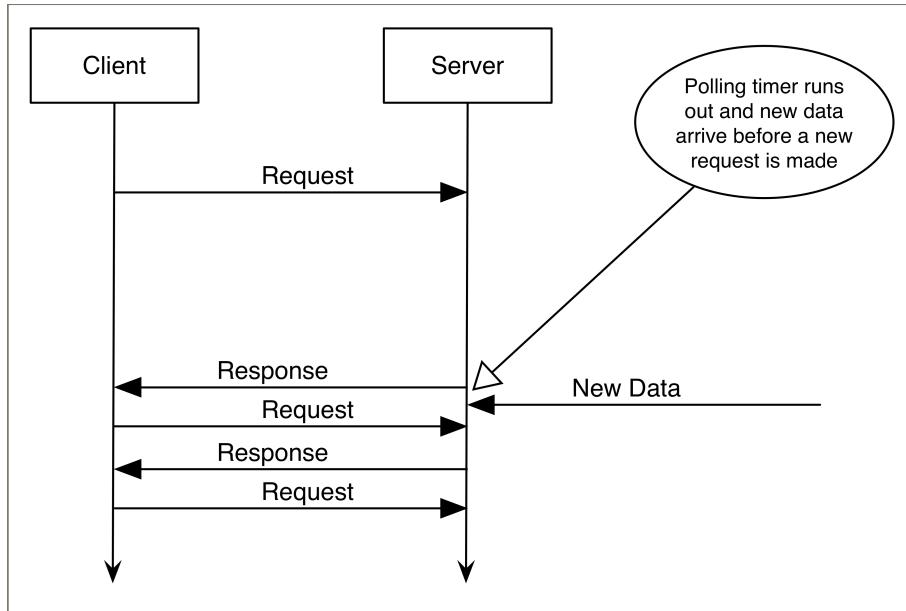


Figure 5: Example of client side outdated data with Long Polling.

Long Polling also faces a problem when there are many updates of data and the client constantly has to reissue a polling connection. At this point Long Polling almost becomes regular Polling.

Another issue is related to HTTP headers. With my GET example to www.uio.no from subsection 2.3.4, the amount of header data in all the requests are between *500 and 800 bytes*, and all 56 response headers are between *300 and 500 bytes*. With many real time applications you only want to send small messages, maybe just a couple of bytes. This vast amount of unnecessary header data is repeated for each packet and could congest the network.

Lastly, let us not forget that all Comet techniques are only for server-to-client messaging. If a real-time web application requires client-to-server messaging as well (i.e. a chat application), one needs to do those with HTTP POST requests. As HTTP is stateless and not connection oriented, server-to-client and client-to-server messages are independent from each other, and the server complexity increases to accommodate for that. All these problems would have been fixed by a connection oriented protocol like TCP.

To sum up, even though Long Polling and HTTP Streaming accomplishes push behavior and near real-time, there are several disadvantages to using the two techniques. Compared to the lower level more powerful TCP sockets, building real time networked applications for the web with HTTP introduces various obstacles.

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[6]. Even though HTML5 adoption started many years ago, the W3C recommendation was just recently finalized[13]. HTML5 is, despite its name, much more than just an updated HTML version. It is a collection of many technologies that are intended to clean up the syntax and unify web technologies. It also introduces many new APIs that makes 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 solves the issues with server-push we saw with Long Polling and Streaming. It is a new feature of HTTP and it provides the concept of a connection. As its name implies, Server Sent Events is unidirectional and supports only server-to-client messaging. To understand Server Sent Events, you must understand the EventSource API[14] 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.

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 provides three events a client can listen for (*open*, *message* and *error*). A simple close method is added for connection teardown. Server Sent Events is built for server-to-client messaging only, so there is no send method, as we see with WebSocket later. The figure below shows how simple it is to open up a connection and listen for messages:

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

Example 3: SSE connection example.

2.6.2 Event Stream Protocol

Under the hood, Server Sent Events is actually implemented as HTTP Streaming over a long lived HTTP connection, but with a consistent and simple API. Other advantage over regular

HTTP Streaming includes automatic reconnects when the connection is dropped and message parsing[15].

Syntactically, what makes Server Sent Events different from HTTP Streaming is the Accept and Content-Type header fields. The new value is “text/event-stream”. To illustrate how this exchange is done and how data is sent, see the following example.

```
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 example.

In the example above 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[15]. It 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 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. Every modern browser except Internet Explorer supports Server Sent Events, but since Internet Explorer accounts for a very large portion of the market, choosing Server Sent Events alienates those users.

While many apps would benefit from server-push with Server Sent Events, some apps require client-to-server messaging as well. Server Sent Events has no answer to that problem (unless used in cooperation with HTTP POST). That is left for WebSocket to solve - the full-duplex TCP like protocol for the web.

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. WebSocket utilizes a single TCP socket and simplifies some rough edges of the underlying protocol, with a simple, powerful and elegant API. WebSocket seems to be the protocol that eradicates the advantage platform native applications had over web applications. It promises to be all about performance, simplicity, standards and HTML5[16] and 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 of the technology; the protocol itself (RFC 6455[17]) and the API.

2.7.1 The WebSocket API

One of the great powers of WebSocket is its simple, yet powerful JavaScript API. The API was defined by the W3C and is the interface you interact with as a web developer. The following figure 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 on. In addition there are *extensions* and *protocol* attributes. What they are will be discussed below. With being full-duplex, there must be a way to send messages as well as receiving. This is done with the send methods. These method accepts either String or binary data. Since this is a new protocol, Strings are expected to be coded in UTF-8, removing all encoding problems. The close method is called when you want to terminate the 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

With the API explained, it's time to look into the protocol itself[17]. The WebSocket protocol was designed to work seamlessly together with HTTP. In fact, you need to have an HTTP connection open before you initiate a WebSocket. When the HTTP connection is up, WebSocket uses a 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 seamless rollout of the protocol. This upgrade protocol is part of what is called the WebSocket Opening Handshake[18].

WebSocket Opening and Closing Handshake

To open a WebSocket connection, a client sends a 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 protocol. 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. Actually, this was a simplification since there is also an exchange of keys going on. The key exchange is there to make sure the two parties talk the exact same protocol version.

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

```
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: HSmr0sMLYUkAGmm50PpG2HaGWk=
```

Example 6: WebSocket opening handshake example.

Message Format

To improve the lives of developers and to keep its simplicity, WebSocket abstracts away some of the roughness of TCP. When you want to send a message over a TCP socket, the message might be divided into several chunks and you have to deal with the fact that they are delivered as chunks and not as whole messages when they arrive. 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 - or frames. A WebSocket frame looks like this:

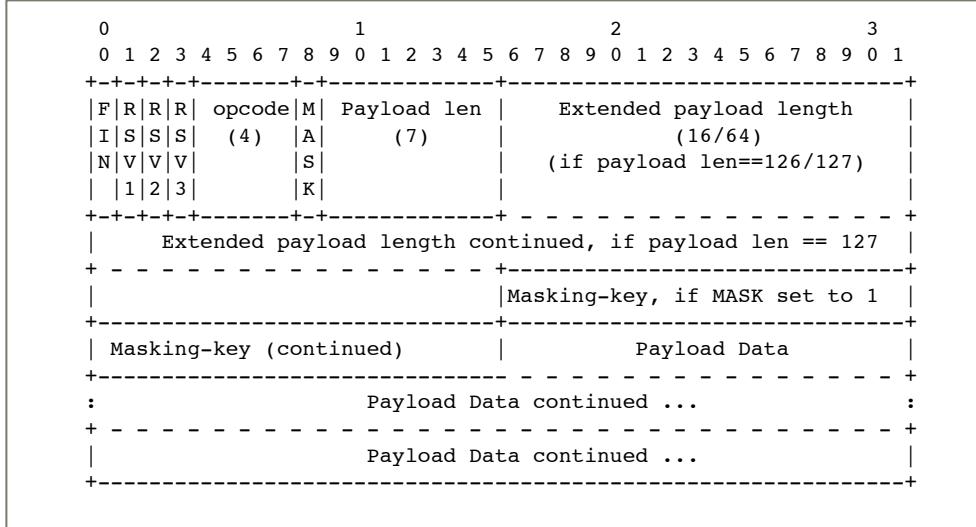


Figure 8: The WebSocket frame.

For this thesis, most of the fields in a WebSocket frame is not that important. But I want to show it anyway, 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 extra 8 bytes (7 + 64 bits) are added. For very small messages then, only 3 bytes of header data is necessary! Compare this to HTTP where each message needs many header fields.

Subprotocols and Extensions

The simple, yet powerful nature of the WebSocket API makes it perfect to build higher level protocols and frameworks on top. This was thought of when WebSocket was designed and the protocol fully support what is known as *subprotocols*. When creating a WebSocket connection you can pass in an array of subprotocol names like this:

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

Example 7: WebSocket connection with subprotocols.

In the above example the client tells the server at example.com upon connection that it speaks both ‘proto1’ and ‘proto2’ and if the server knows these, the server can chose which one to use, but only one at a time. There are several official protocols[19], such as Microsoft SOAP and unofficial open protocols such as XMPP. It is possible for anyone to create additional WebSocket subprotocols.

In addition to subprotocols, there is another way to supplement WebSocket with additional features: WebSocket extensions. Unlike subprotocols you can extend your WebSocket connection with several extensions. An extension is a supplement to the already existing protocol and both browser and server must support it. Extensions can be added with the *Sec-WebSocket-Extension* header and following is an example that compress frames at source and decompress at destination:

Sec-WebSocket-Extensions: deflate-frame

Example 8: WebSocket extension.

2.7.3 WebSockets vs. HTTP

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

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

One of the issues with HTTP, was the large amount of header data. With my HTTP GET example to www.uio.no, every request and response had several hundred bytes of meta data. Since WebSockets are stateful, message size can be tiny in comparison (but requires more action at the server). To illustrate the difference in message size and header data, I have created an example based on the stock price application example from section 1.1. How does HTTP Long Polling, Server Sent Events and WebSocket solve server-push? The stock price update is commonly 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:

```
{  
  "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 additional 221 bytes are used for HTTP headers, totaling at 279 bytes per stock price update. 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, let us not forget that since HTTP is half-duplex, every HTTP response follows a HTTP request. Assume that each HTTP request looks like this 263 byte long example:

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 are no need for a HTTP-like request, once the connection is up. There are not much wasted space on headers either. The message that is pushed from the server can look like this:

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

Example 12: Server Sent Events message example.

Including a blank line at the end, each stock price message totals at *74 bytes* with Server Sent Events. 16 bytes of header data is not bad considering how inefficient HTTP was.

WebSocket

The stock price message above is 58 bytes long, well below 126 bytes, meaning it requires only 3 bytes of header data with WebSocket (see section 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 it is binary encoded compared to the ASCII text based HTTP and Server Sent Events.

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 front and backend. Unlike the way it is for OS native applications, the web front end is limited when it comes to development choices. Your code have to be JavaScript, HTML and CSS. This is not entirely true as shown with newer languages like Dart[20], CoffeeScript[21] and TypeScript[22], that acts as replacements for JavaScript. Still, the browser do not understand these languages, so they ultimately need to be compiled to JavaScript. JavaScript is in a sense the assembly language of the Web.

On the backend however, you are free to chose your preferred web framework and language. Traditionally Java and .NET with frameworks such as Spring and ASP.NET respectively, have been very popular. Even though the clear separation of front- and backend works fine, a newer platform called Node.js shows there was a need for a more unified web development process.

2.8.2 Node.js

As web applications became more complex following with Ajax, web developers spent increasingly more time writing rich client side applications in JavaScript. The context switch from frontend JavaScript to the backend with another other programming language could be cumbersome. So when the creator of Node.js Ryan Dahl introduced server side JavaScript[23] in 2009, many developers found the promise of JavaScript everywhere promising.

Node.js is a JavaScript runtime environment built upon Google Chrome's V8 JavaScript engine. As V8 is mostly written i C++[24], it can run directly on the hardware and is as a result, really fast.

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

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

In traditional threaded web servers, a new thread is spawned for each new connected client and the server context switches between all threads and runs their code. However, most of the time, web servers are doing IO, typically querying a database or reading a file. IO operations block the running thread and the server and have to wait for the IO operation to complete. 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, along with the fact that programming for a threaded environment is hard.

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 on 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[25].

To show how the two different programming styles are, consider the following examples:

```
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 the first example 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 programming is done in a threaded and synchronous environment. Most programming languages like Java follow this model.

The second example 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 asynchronously manner is fundamentally different to the synchronous style most back end programmers are used to with Java. 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 browser already has an event loop constantly listening for events such as button clicks. Node.js unifies web development around only one programming style and language.

NPM (Node Package Manager) is another great feature of Node.js. NPM makes it really easy to install packages for you to use in your projects. It works like this:

```
$ 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. This means that memory allocation and deallocation is 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 memory leaks are then less likely to happen.

However, a garbage collector introduces a performance penalty. V8 uses the stop-the-world collection scheme[26]. This means that V8 stops all program execution once the garbage collector runs. As a result, the CPU and response time will be affected for the time the collector runs. This makes the performance to a certain degree less predictable, as the garbage collector can decide to run at unpredictable times.

There are of tools to memory profile your Node.js application, but since Node.js is a new platform, the tools are not as mature as they are in other environments, like the JVM.

2.9 Performance Testing

2.9.1 Introduction to Performance Testing

To determine what technology is the most efficient under a set of certain 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”[27]. 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 a integral part of software system 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, making a stressing situation for the system. This can include limited memory or processor resources.

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

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

In this thesis however, when performance testing WebSocket to Server Sent Events and Long Polling, I will only focus on *load* and *stress testing*. For simplicity's sake, I have defined Load and Stress testing to mean the following in this thesis:

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

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

The figure below shows how a load test “becomes” a stress test once the CPU load reaches maximum and plateaus out.

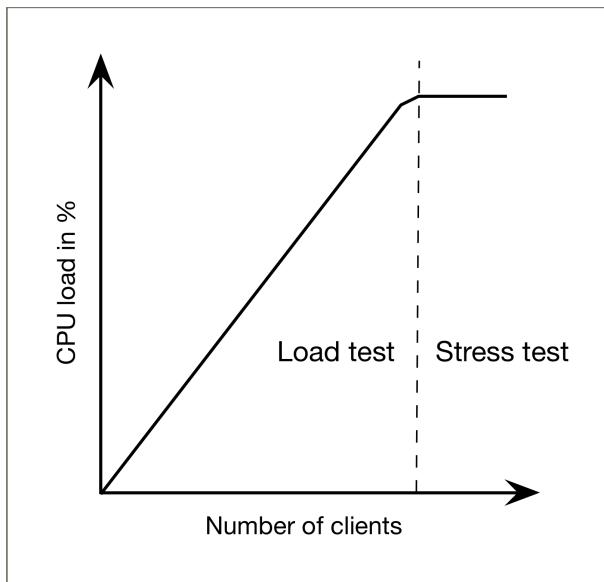


Figure 9: Relation between load and stress tests.

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[28] written by Jakob Nielsen and is an excerpt of 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, 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 time fairly low. However, as the server reaches maximum CPU load, I expect the response time to slowly grow linearly with the client count. The figure below shows how the expectation looks like:

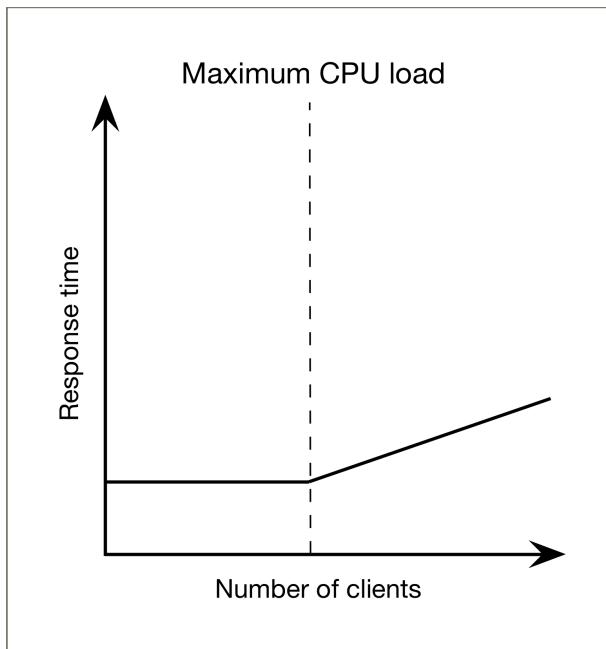


Figure 10: Expected Response Times.

After reading the related academic work and learning about the three different transports, I have formed an expectation that WebSocket is the transport that will 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, and because WebSocket provide a small overhead in terms of headers, 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 have 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 be the worst performer is the fact that there must be a request for each server-pushed message. This 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 introduces connections. These connections will consume memory in a higher degree than the incoming HTTP requests. As WebSocket is more advanced (full-duplex) than Server Sent Events, I expect it to be the most memory hungry of them all.

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 two first sub-questions are related to performance, while the third and last is related to programmer friendliness and ease of use.

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

The thesis introduction presented two types of real-time applications - the stock price example as a unidirectional messaging application and the chat application as 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.

The first test scenario is a broadcast application where a given number of clients connects to a server. This server receives incoming messages from an independent backend component. When the server receives these messages, it immediately broadcasts them to the connected clients. In a sense, this is the stock price application.

The second test scenarios is a chat application. It consists of a given number of client that connect to a server. Each client periodically sends chat messages to the server. The server then broadcasts these messages to the clients, as they come in. Naturally, with HTTP being unidirectional, the Server Sent Events and Long Polling versions, requires an additional client-to-server component in this scenario.

3.1 Test Scenarios

This section gives a quick description of the two test scenarios and what components they consist of. 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 and the text describing this scenario will not distinguish between the different version. 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 immediately broadcast these to all the connected clients. You can think of this system as the stock price app example from earlier. In a real world scenario, the clients would be web browsers.

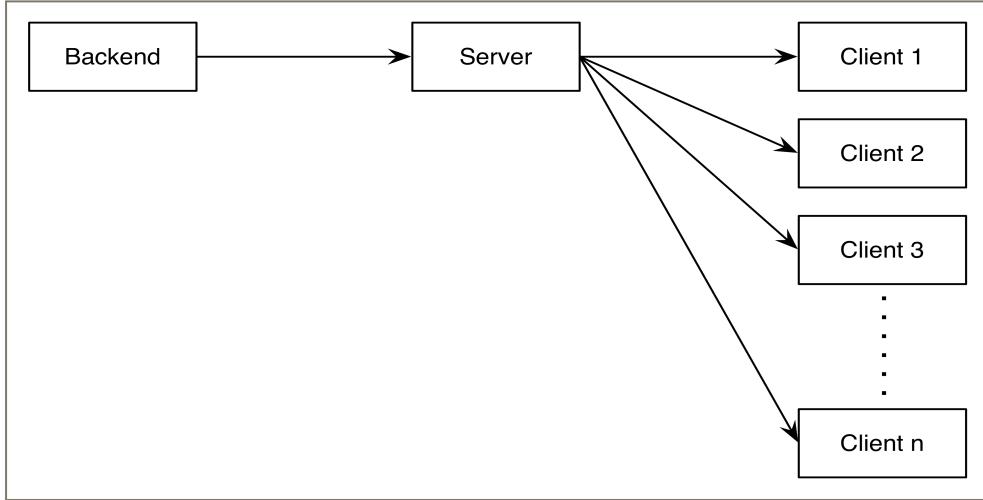


Figure 11: The three components in the first test scenario.

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) and the text describing this scenario will not distinguish between the different version. 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 either WebSocket, Server Sent Events or Long Polling. After some initial exchange of information, the test is started. 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. A simple figure showing the components can be found below.

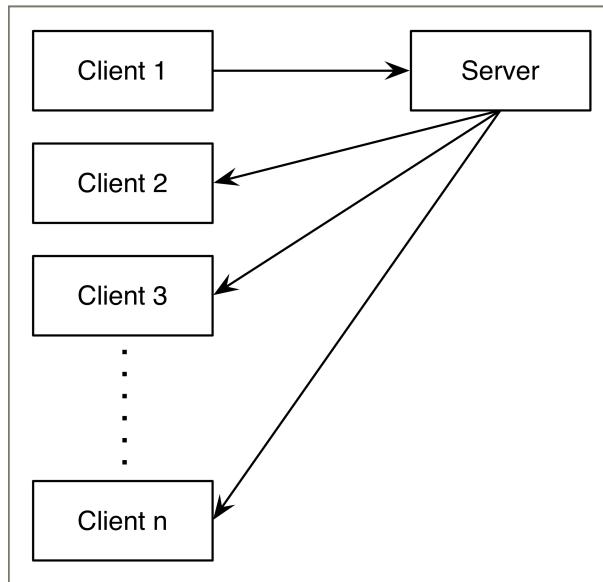


Figure 12: The components in the second test scenario.

3.2 Test Data

The main research questions state that the purpose of this thesis is to compare WebSocket to other real-time technologies for the web and try to answer what types of applications that would benefit from WebSocket. To get an accurate picture, I have chosen to load and stress test the different transports. Stress testing the 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 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 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 is *CPU load* and *memory footprint*.

The other perspective is the user's. 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) from the client side.

3.2.2 Collection Through Three Test Phases

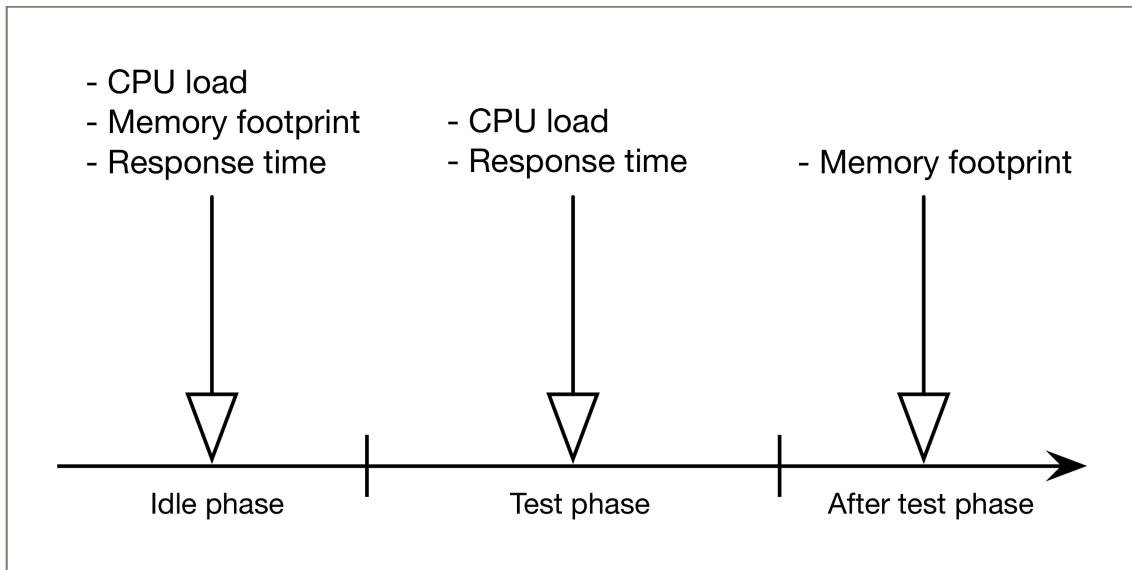


Figure 13: The three test phases and what is measured in each phase.

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 (or polling) to the server and everything is set for the test 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. In the second test scenario, this is the phase when the chat is live. The memory footprint will gradually increase throughout this phase as the server receives more messages. But, the garbage collector could clean and free memory space as well. When this happens and how it affects memory is not easy to say. Because of this uncertainty and unpredictability in terms of memory use, the memory footprint is not collected during this phase. Only the CPU load and response time are collected.

The last phase is the phase immediately after the test have finished running. CPU and response time will be the same in this phase as in the idle phase, as the clients are inactive (or disconnected), but the memory consumption would not. As stated in the paragraph above, there are some uncertainties with regards to memory, but I decided to collect memory footprint after the test, to make sure I spot any (if present) irregularities.

3.2.3 Collection

The server side metrics (CPU load and memory footprint) are collected by a separate monitoring process running on the server machine. Every 50th millisecond the CPU load and memory footprint is recorded. The server process notifies the monitoring process when the different phases are started and it is the monitoring process' responsibility to calculate the CPU load and memory footprint average when the test is finished. This average is sent to the server for print out on the screen.

The client side metric, server response time, is collected differently between the two test scenarios. In the first scenario, there is a separate ping client that every 50th millisecond sends a timestamped 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

When performance testing the server, it is important that it is isolated from all the other components. 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. Isolated process running on same hardware as clients and backend.
2. Isolated virtual machine running on same hardware as clients and backend.

3. Isolated online server instances from an online cloud provider.
4. Isolated on a different physical machine running in an isolated local network.

The first alternative is ideal for development as everything is run on a single computer. For testing however, it is not. It is difficult to tell how the OS context switches between processes and how much time it actually uses on the server. It would be better if the server software was the only process, except for the OS, running. Also since this is about testing network protocols, it is not a good idea to run the clients and server on the same machine. To get an as accurate picture of the server load as possible, the server should be isolated on a hardware level. 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 hardware the server runs on. 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), because the server must reach its resource limit before the client machine for this to be a server 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 OS 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 - protocols. But, benchmarking protocols is not really possible as a protocol is just a set of rules. Implementations of protocols, on the other hand, is possible to benchmark.

To get the most accurate picture of how WebSocket compares to Server Sent Events and Long Polling, I would have to compare each and 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 could have chosen the most popular real-time frameworks and compared them, but that is essentially what Johannessen[2] did.

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[25]. 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 can make development fast, especially with tools like nodemon[29].

Great performance and programmer friendliness makes Node.js great 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 monitoring 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[30], Node.js is the web framework that by far is most popular. 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 great compatibility with many existing projects.

Node.js comes with the great package manager NPM[31], that makes it really simple to quickly 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 the language 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 to only have 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, since 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 project includes some web components. With the latest edition of OS X by Apple, there is even a JavaScript interface to the OS[32]. Also, I chose it so that I can learn some of its quirks[33], as I'm likely to work on some web project in the future.

Node.js is perfect for creating command line programs

Node.js has great support for creating command line utilities with the readline module[34]. This makes it perfect for the lightweight clients.

3.3.3 Command Line Clients

In his Further Work section, Kristian Johannessen[2] suggested using smaller lightweight console applications or headless browsers as clients. As the purpose of the tests in this thesis is to compare transport technologies at scale, it is preferable not to use full blown web browsers as clients. Real web browser clients would consume quite a lot of system resources. Self-written console apps on the other hand gives full control and let me put my focus on what I want - the transports.

3.4 Test Configurations

As the tests were designed to be load and stress tests, they had to be run in such a way that the server reaches its resource limit with the chosen hardware. In this section, I will present and discuss the parameters for both test scenarios that 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 number of clients for the tests. As a default, OS X allowed 709 user processes. On the client machine, about 220 user processes was constantly running. To then spawn 500 client processes was not possible without increasing the OS level maximum process limit. I increased the limit to 1024 with the following commands:

```
$ 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 possibly have had more clients, possibly 600, but then OS X would sometimes freeze and tell me that I have too many user processes, event 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 transports 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 5 th 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[35]. 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. With 5 milliseconds between each message, this means 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. I wanted to reach the break point in response time 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 about 25 seconds. That made it run long enough for an accurate picture, but at the same time not too long and making it unfeasible. I chose 30 seconds for each test.

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. The following figure shows an example with five clients sending their first two messages.

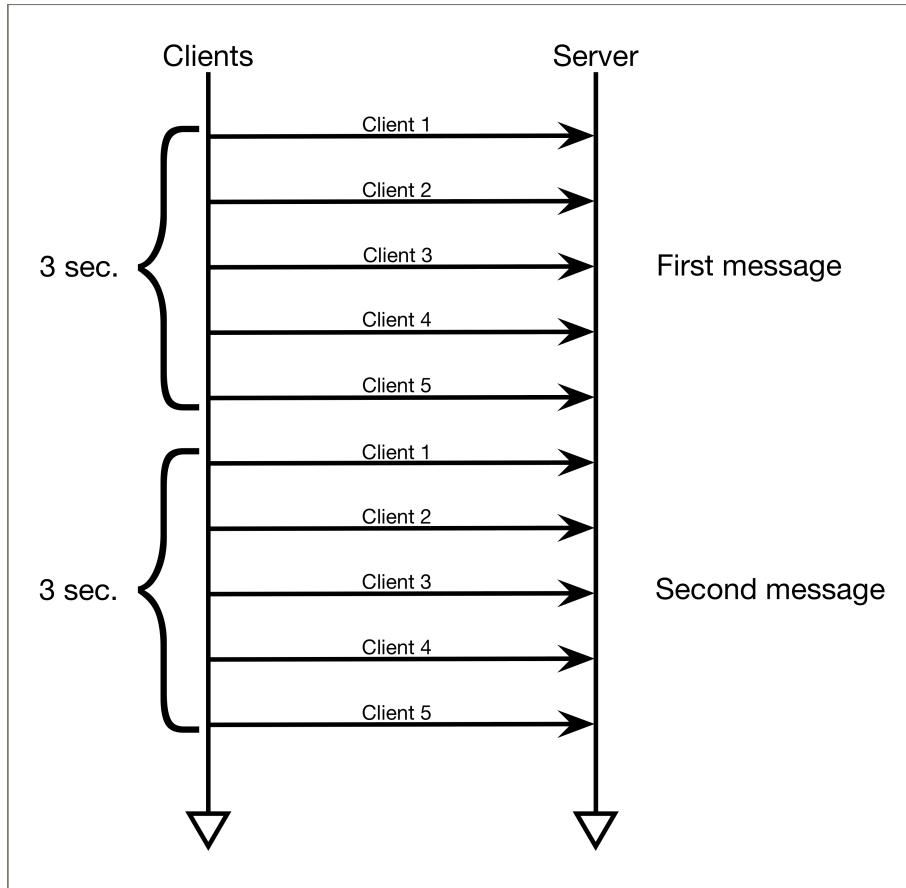


Figure 14: Example showing five clients sending their first two messages.

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 response time.
- Monitoring process: The process responsible for measuring CPU load and memory footprint on the server is a process spawned by the server.

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. The figure below shows how the information flow is during a test.

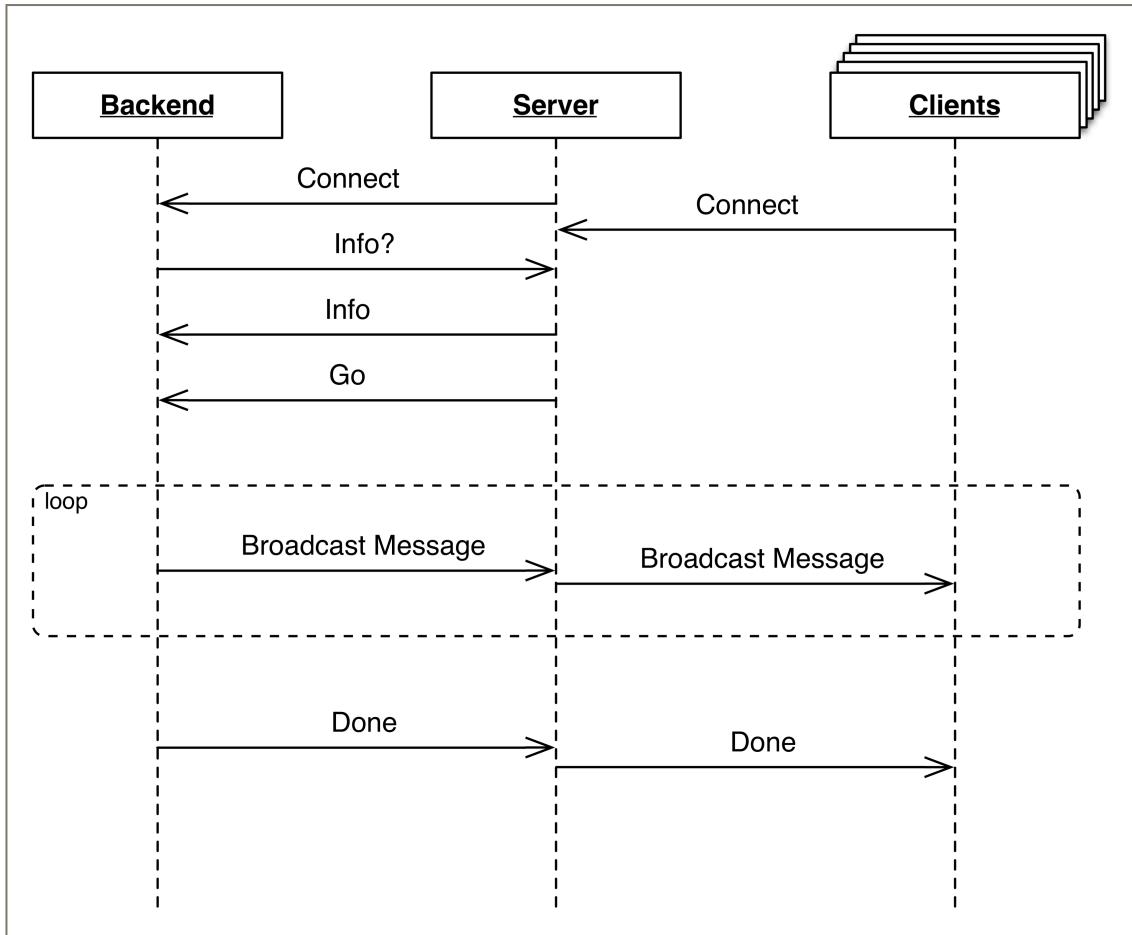


Figure 15: Sequence diagram for the first test scenario.

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 a message 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 message stream. 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 that the broadcast is over, and calculates the average CPU and memory usage before and during the broadcast. This is sent to the server that lastly 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 dead 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 is received, the client reports to the master client that the broadcast is finished and reports whether it received all messages.

The ping client is a process that every 50th milliseconds sends a message with a timestamps to the server. The server instantly sends this message back and the ping client calculates the time it took to get a response. When the ping client pings the server after the broadcast is over, the server replies with a done message and the ping client 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:

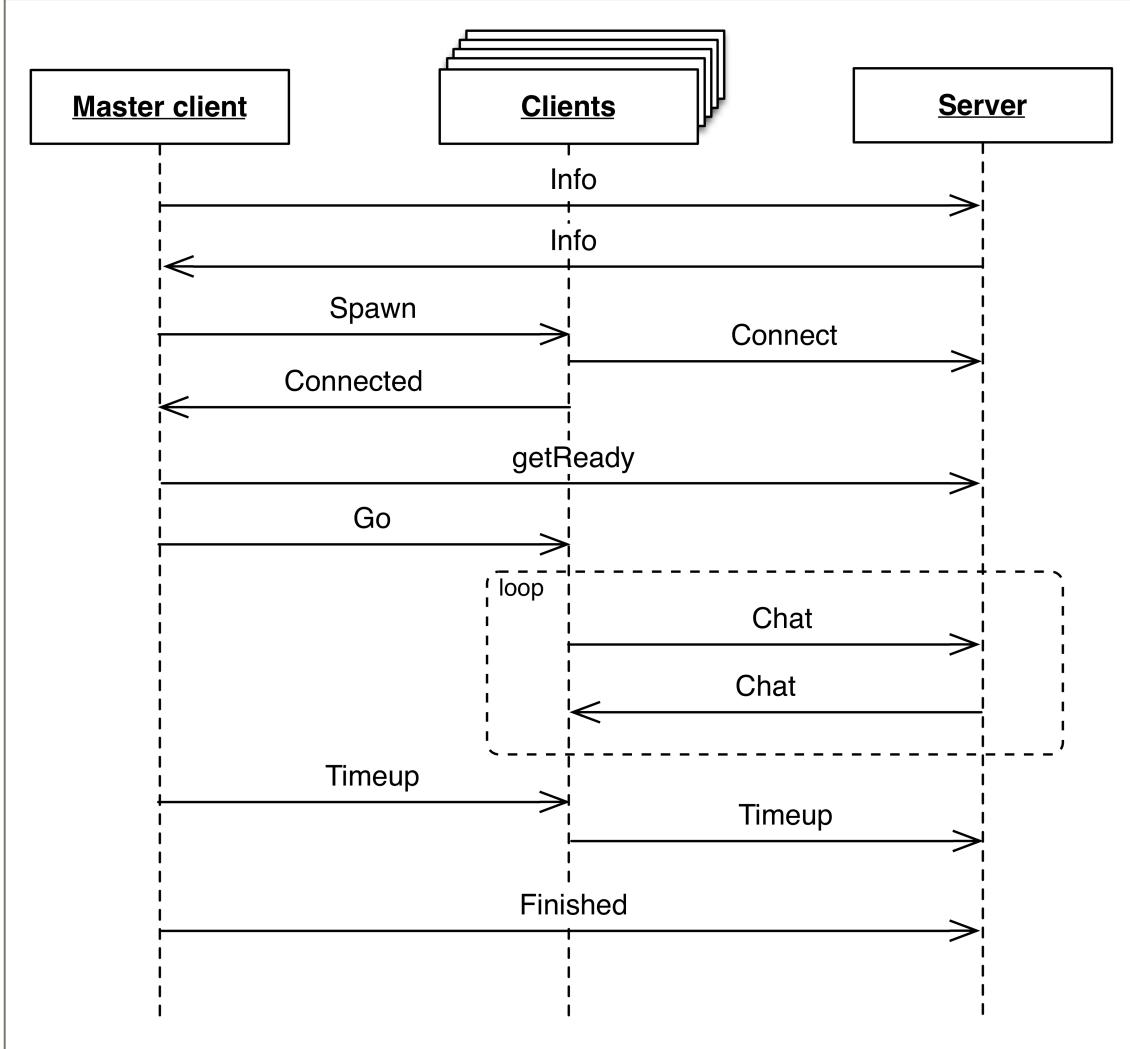


Figure 16: Sequence diagram for the second test scenario.

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 timestamped when sent, so that the clients can calculate response time themselves when they receive a chat message.

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

number of messages have been sent and the client makes sure 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

When the master client then connects to the server, some information is quickly exchanged, so that both parties know how long the tests should run and how many clients are involved. The server then waits for a *getReady* message indicating that the test is about start. When the *getReady* message is received, the server forks a monitoring process so that it is ready to start measure server resources when the chat test is starting.

When the first *chat* message has been delivered to the server, the server tells the monitor to start monitoring server resources. For every chat message that arrives, the server immediately broadcasts it to all the connected or polling clients.

When the chat phase is finished, the server receives *timeup* messages from the clients, meaning that the chat is finished. The server then tells the monitor to stop monitoring. 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, going over the three versions (one for each transport) for the first scenario and the three versions for the second scenario. The two test scenarios share a lot of common code and libraries:

3.6.1 Common Between Scenarios

As recommended by Kristian Johannessen, 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 put the focus on the transports. By choosing to do all tests on a single platform using small, fast libraries, and lightweight console clients, the focus could stay on the technologies in question.

WebSocket

There are no official client or server implementation 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[36]. 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[37] 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[38] WebSocket implementations, regardless of 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 implementation of SSE for Node.js, either as a server or as a client. On the client side the choice fell on *EventSource* by Aslak Hellesøy[39]. The library is small and doesn't 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. This is essentially all needed for a HTTP server to become Server Sent Events-ready.

```
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*[40] helped to make that a reality. In addition, the small library *request*[41] 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[42] was used. It provides a simple interface to get CPU and memory usage of a process, using the UNIX tool *ps*.

At first, the CPU and memory monitoring was included into the server process itself, but as the CPU load increased, it started giving irregular and incorrect results. After investigation I learned it was because of Node.js' Event Loop. The monitoring events got lower priority than the broadcast events and eventually never got run as the server always got new broadcast messages to distribute. Consequently it was separated into its own process, forked by the server process.

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, so 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, just one component for the backend communication and one for the client communication:

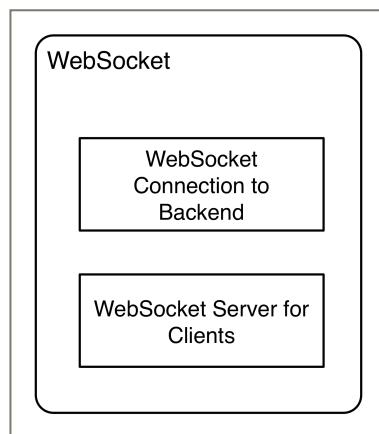


Figure 17: The WebSocket server in scenario 1.

The Server Sent Events version

Once again, the benefits of having a connection oriented transport, makes the server development a joy. The Server Sent Events server is very similar to the WebSocket counterpart and works the same way:

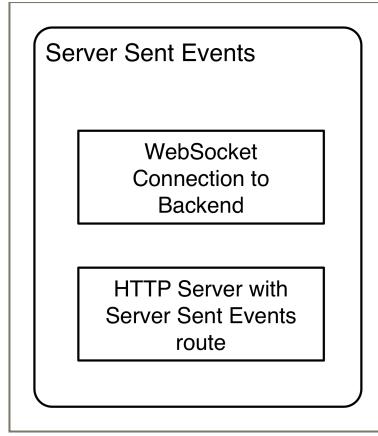


Figure 18: The Server Sent Events server in scenario 1.

The HTTP Long Polling version

HTTP has no concept of persistent connection, so this server needed to be a bit more complex. See subsection 2.5.3 to understand why this server needs to store each and every message that arrives from the backend. The figure below shows how much more complex this server is:

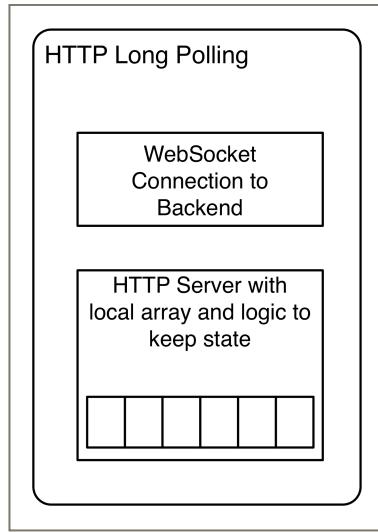


Figure 19: The HTTP Long Polling server in scenario 1.

Ping Client

The ping client is forked by the client starter process and constantly (every 50th millisecond) 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 SSE and HTTP Long Polling, its 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, as previously stated, HTTP is not full-duplex and Server Sent Events is, as the name states, only for messages going server-to-client. To allow the clients to send chat messages to the server, traditional HTTP POST messages was used. In a sense, the Long Polling version is Long Polling + HTTP POST, while 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 server when I refer to the different ones.

The client spread and message frequency

As touched upon in the Test Configuration section above, 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. The figure below shows that both incoming and outgoing messages goes 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.

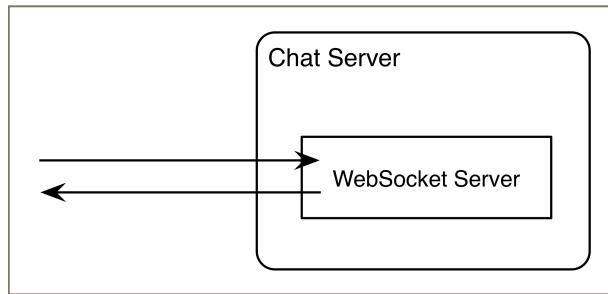


Figure 20: The WebSocket server in scenario 2.

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 over Server Sent Events. With Server Sent Events being connection oriented, there was no need to store messages on the server side.

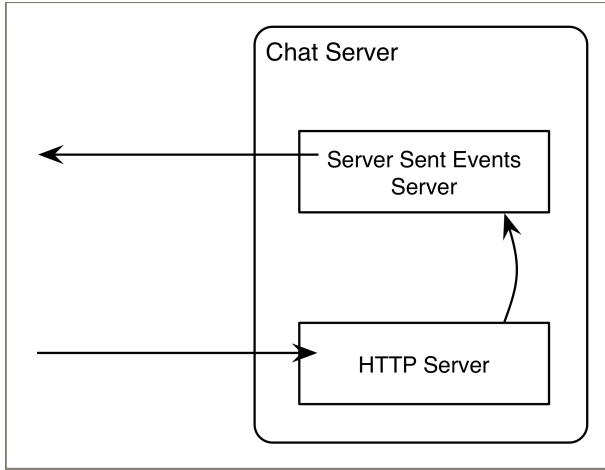


Figure 21: The Server Sent Events server in scenario 2.

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, there is no concept of “connected clients” as they “lose” 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. The figure below shows the complexity of the implementation.

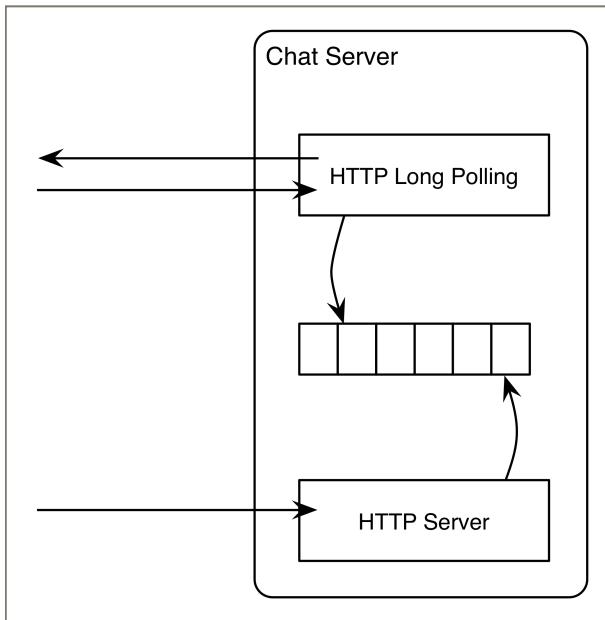


Figure 22: The HTTP Long Polling server in scenario 2.

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, possibly several days for a single test. That would have been another thesis in itself.

3.7.2 Network Use

In the background chapter, 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 break point 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[43]. I do not believe my tests are written in an incorrect or error prone way. The reason I believe this is that the test results from the first scenario are comparable, and similar in some ways, to the results from the second scenario (as seen in the following two chapters).

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 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, since a thesis is time constrained and with recommendation by Johannessen[2] in his thesis's Further Work section, 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

As discussed previously 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 that needs to be understood. I briefly presented the Event Loop in the background chapter, but did not mention the following weakness. When the Event Loop has triggered and called a callback, it is blocked. Usually not for long, but if it is stuck on a computationally heavy task, it can cause slow response times or connect issues[44]. With being an interpreted and dynamic language, JavaScript can not be optimized

in the same way as i.e. 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 response times. Another problem with the garbage collector, 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 with regards to the garbage collector, I did not want to focus too much on memory with my results.

Chapter 4: Test Results

This chapter presents the results from the two test scenarios. Each of the two tests were run ten times for a given number of clients for each of the transports. The client count goes from 1 to 500 with increments of 50. This means 330 test runs (10 test runs * 11 different client counts * 3 different transports) for each of the two test scenarios, resulting in a total of 660 test runs. The average of the ten test runs is calculated. It is this average that is used to draw the charts in this chapter. For the complete test result records, see the Appendix.

As stated in subsection 3.2.2, the tests have three different phases, as the illustration below shows. The results 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.

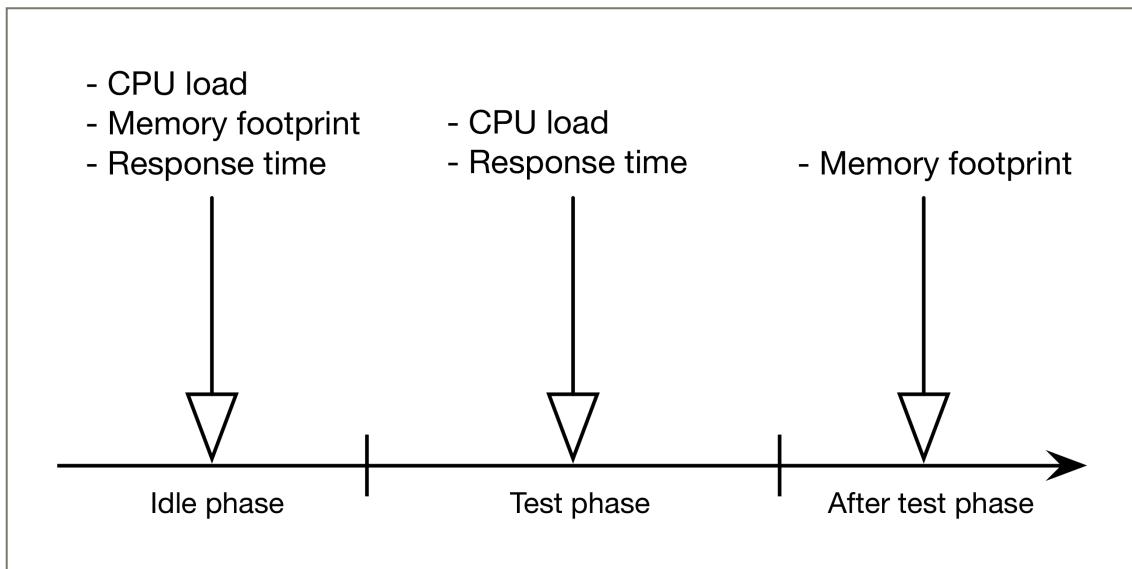


Figure 23: The three test phases and what is measured in each phase.

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 - idle. Idle clients should consume as little server resources as possible enabling a short response time for clients.

In this phase, I do not distinguish between the the two test scenarios, as they are similar until the test phase.

4.1.1 CPU Load

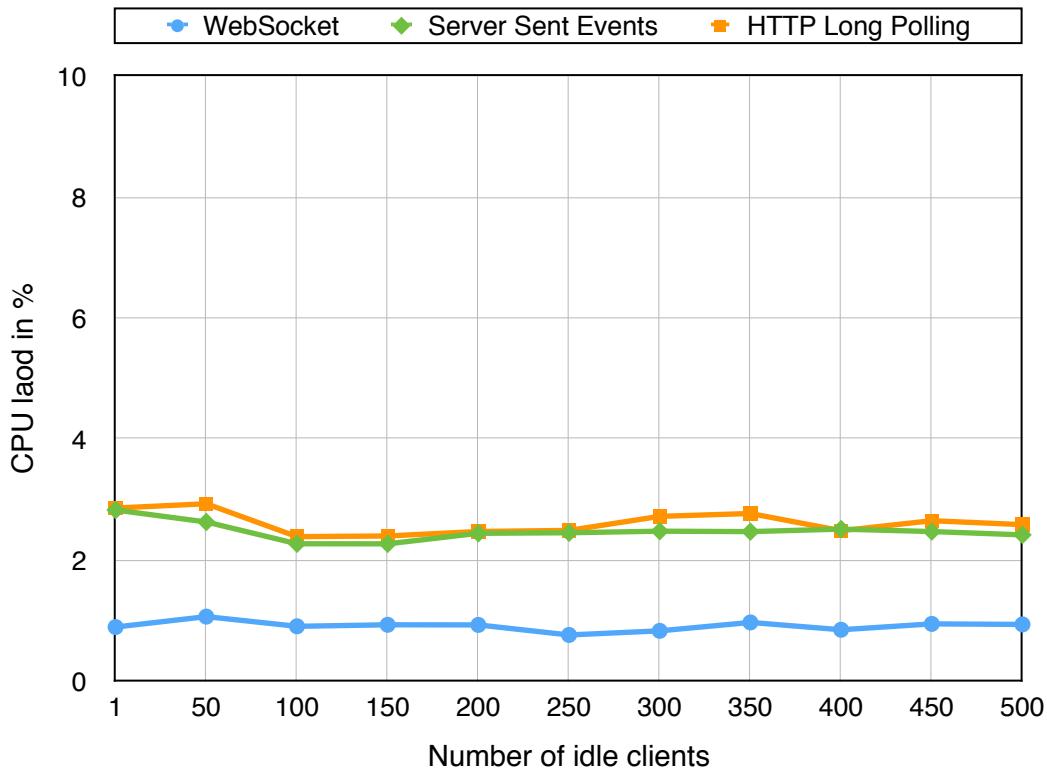


Figure 24: The CPU load for all three transports during the idle phase.

All three servers used so few CPU cycles that the Y-scale had to be set to a maximum of 10% to show the difference between them. 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 cycles and stays around just 1%.

Even as the client count rises all the way up to 500, the CPU usage stays low and seem totally 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[40] for HTTP support.

These results are good results, showing that all three servers support 500 inactive clients with no issues related to CPU use. The WebSocket server is in a league of its own, but the results from the other two servers must also be considered good.

Even though these results were good, they were expected (see section 2.10). Once a client connects to a server and stays inactive, there is no work for the server CPU to do. The only place where I expected idle clients to have an effect, was in memory consumption.

4.2.2 Memory Footprint

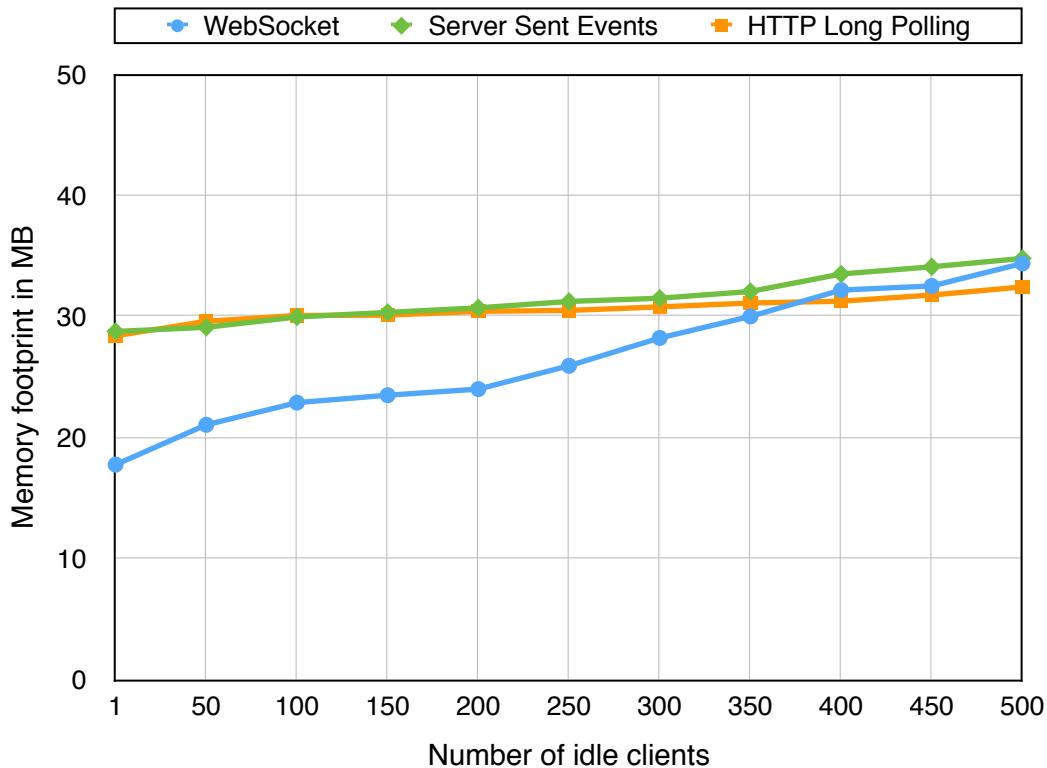


Figure 25: The memory footprint for all three transports during the idle phase.

The HTTP Long Polling and Server Sent Events servers both start off just shy of 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 is 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 though, they start to differ because a Server Sent Events connection is taking up more space than a hanging Long Polling request on the server side.

These results clearly show that a HTTP Long Polling hanging 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 show that these results were expected.

4.2.3 Response Time

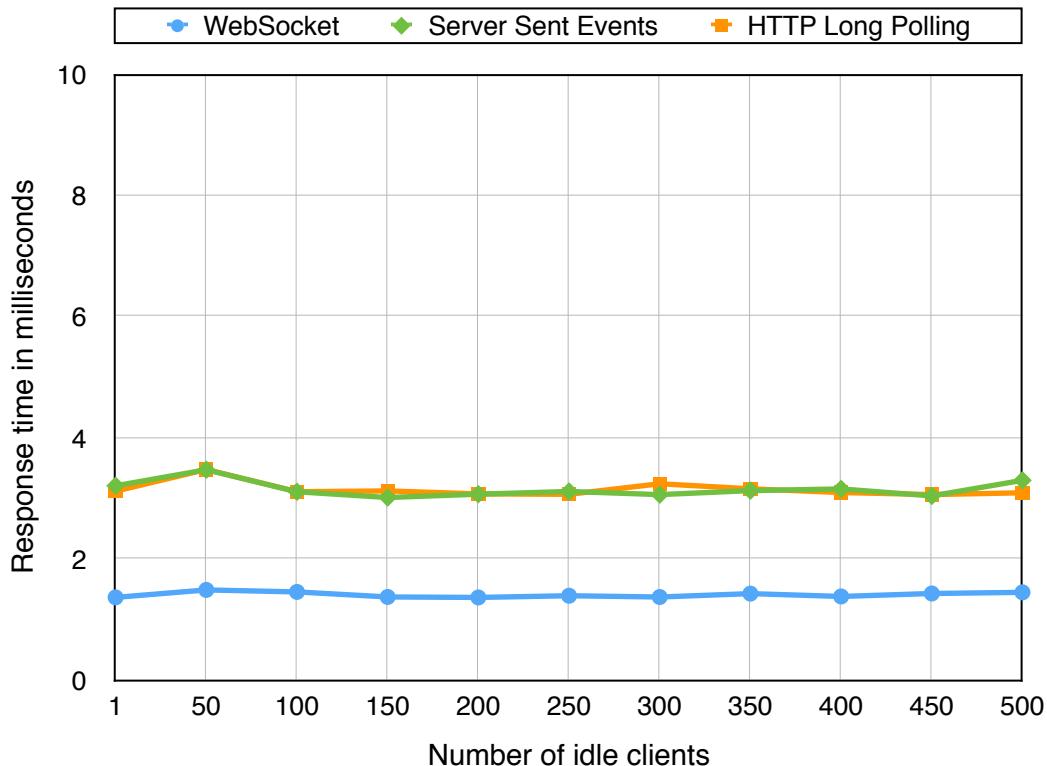


Figure 26: The response times for all three transports during the idle phase.

Figure 33 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 from subsection 2.9.2. Even more impressive is the WebSocket server, with response times always below 2 milliseconds.

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

The almost identical results between the Long Polling and the Server Sent Events servers can again be explained by two factors. First, they both use a HTTP ping route by the same HTTP library. Second, it is the same ping client used in both cases. The WebSocket version, on the other hand, has a WebSocket based ping client.

In section 2.10 I presented my expectation with regards to response times. Because of HTTP header processing and WebSocket being designed for performance, these results were expected.

4.3 Test Phase - Scenario 1

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

The test phase is the most interesting phase as it is aimed towards answering 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 plateaus, the test is a stress test.

4.3.1 CPU Load During Broadcast

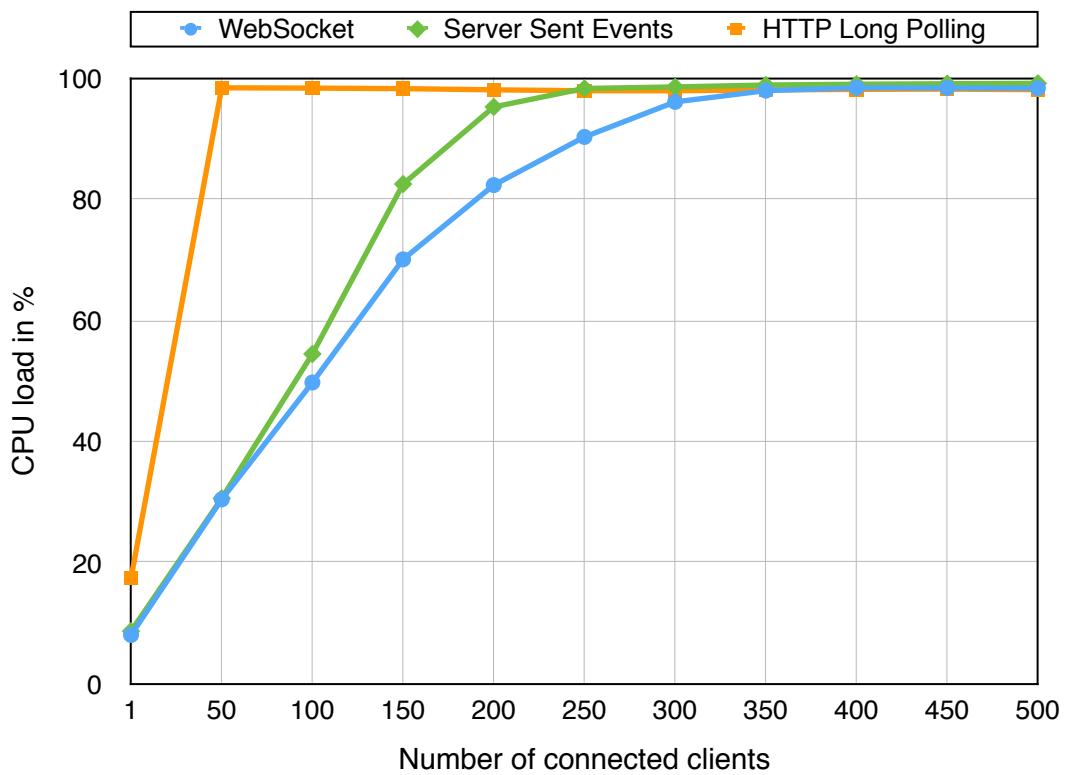


Figure 27: The CPU load during the first test scenario's test phase.

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

These results are in line with my expectations from section 2.10, as the HTTP Long Polling server has to handle more header processing and incoming requests.

These results show that, from a server CPU load perspective, Server Sent Events and WebSocket are clearly more equipped for doing server-to-client real-time messaging than plain HTTP.

4.3.2 Response Time During Broadcast

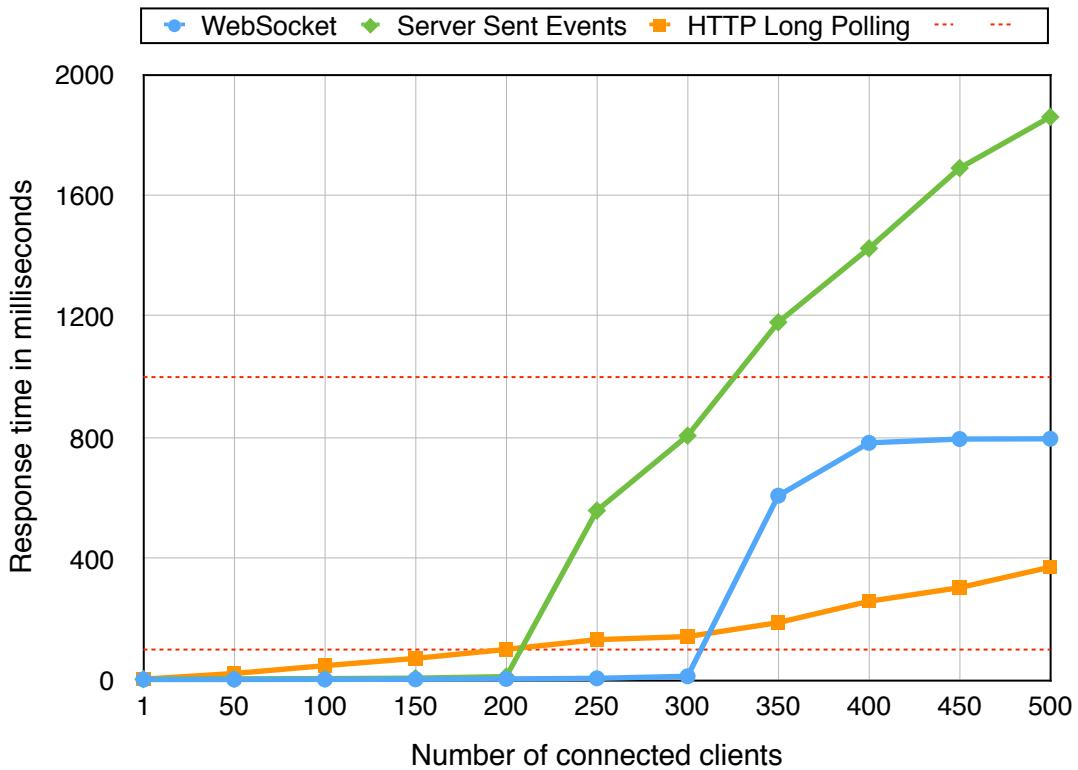


Figure 28: The response times during the first test scenario's test phase.

The two dotted red lines in the figure above, is the 1.0 and 0.1 second limits from section 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 use more than 100 milliseconds to answer a request. At the end, with 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 maximum. From 1 to 200 clients, it always responds within 11 milliseconds, which is well clear of the 0.1 second limit. But as the CPU reaches maximum load at 250 clients, the server response time start to increase dramatically. It immediately breaches the 0.1 limit and at 350 clients, it has even surpassed 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 Server Sent Events server sees an unexpectedly steep climb than the Long Polling counterpart.

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 and matches my expectations from section 2.10 about how well WebSocket fits this kind of application.

With 350 clients, the test becomes a stress test as the server CPU is fully utilized. Now the response time jumps well above the 0.1 second limit. Unlike the Server Sent Events server, the WebSocket server plateaus at around 800 milliseconds with 400 clients.

Once again, this sudden and explosive growth in response time, is totally unexpected. For the response time in the first test scenario, it is only the HTTP Long Polling server that produce expected results. As soon as the server is CPU constraint, the Server Sent Events and WebSocket server really struggle to keep up with the technically more complex HTTP Long Polling. This is not only unexpected, but also surprising.

4.4 Test Phase - Scenario 2

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

4.4.1 CPU Load During Chat

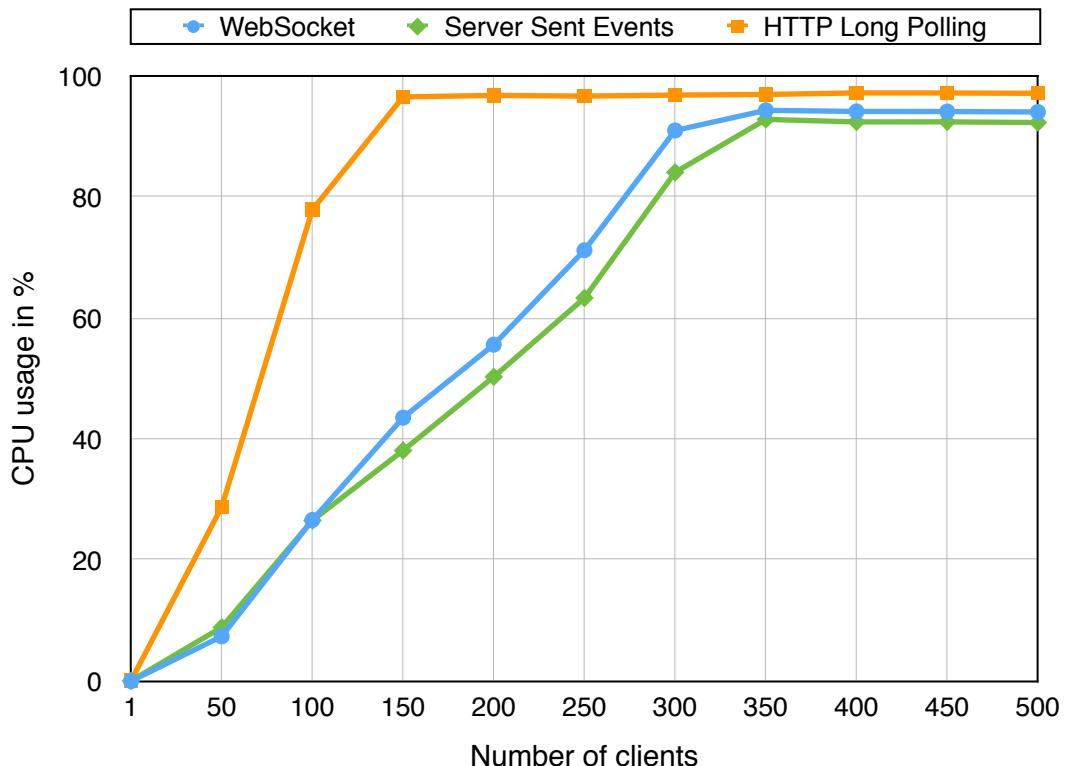


Figure 29: The CPU load during the seconds test scenario's test phase.

The Long Polling server reaches maximum CPU utilization with 150 clients, while the other server reach their maximum at 350 clients both. That Long Polling reaches its maximum first, is in line with the expectations from section 2.10.

That Server Sent Events use less CPU than WebSocket for the same task with the same client count, was not expected. They both reach maximum CPU load at 350 clients, but we see that the Server Sent Events server uses considerable less CPU power in most of the results.

At this point, it does not look like the increase in complexity for the Server Sent Events server has compromised CPU performance compared to WebSocket.

4.4.2 Response Time During Chat

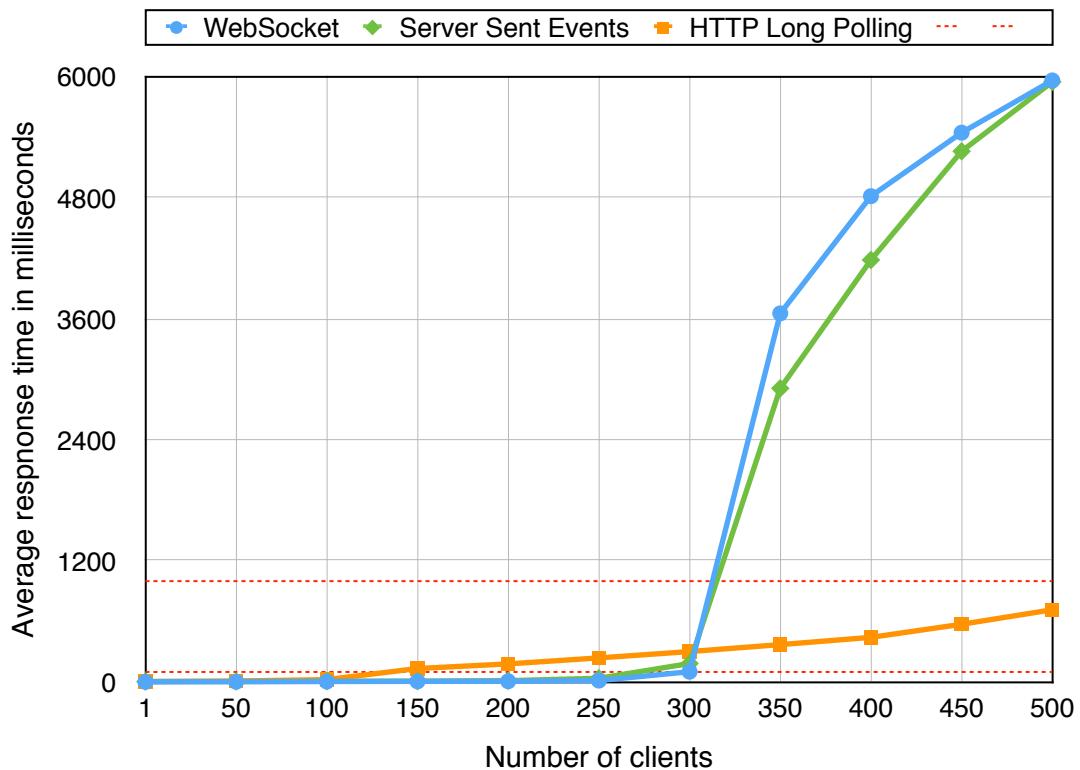


Figure 30: The response times during the second test scenario's test phase.

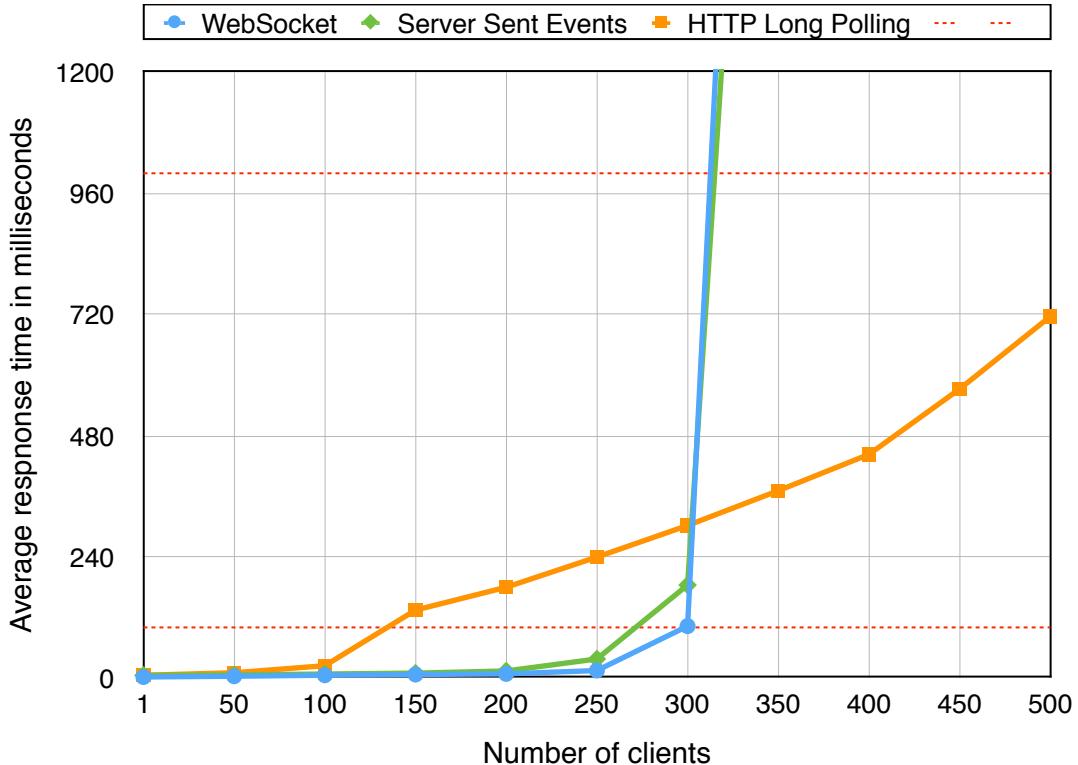


Figure 31: The response times during the second test scenario’s test phase.

Both figures above show the response times from the test phase in the scenario. Because the response time results goes all the way up to 6 seconds, there is a second figure to make the results easier to read, before the dramatic response time increase.

Again, the two dotted red lines in the figures above indicate the 0.1 and 1.0 second limits from section 2.9.2.

When the Long Polling server reaches full CPU utilization at 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 however, the response times breaks the 0.1 second limit.

The Server Sent Events and the WebSocket server performed very similar in this scenario. They both reach maximum CPU load at 350 clients, but already at 300 we see that the CPU is pushed hard enough for the response time to jump quite a lot. 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 growth in response time. With the maximum of 500 clients, both servers have response times of 6 seconds. This is way above the 1.0 second limit.

The fact that the Long Polling server reaches full CPU utilization before the other two servers was expected, as a result of Long Polling being technically more demanding (see subsection

2.7.3). It was also expected that the server would have an slow linear growth in response time when the CPU is stressed.

As long as the CPU load is moderate on the Server Sent Events and the WebSocket servers, they behave as expected, with low response times beating the Long Polling counterpart. But as soon as they are under stressing levels of load, they show unexpected high response times. Even though they were unexpected, they were at least in line with the results from subsection 4.3.2.

4.5 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 focus on memory in this thesis. But, I decided to include these recordings as they produce interesting and unexpected results.

Following in figure 33 and 34, you can see the memory consumption right after the tests have finished. In the first scenario depicted in figure 40, you can see that when the Server Sent Events and WebSocket servers reaches full CPU utilization with 250 and 350 clients respectively, the memory footprint increases dramatically. Both consume well over 1 GB of memory with 500 clients. The expected results would be lower and along the line of the Long Polling server, that lands on 97 MB, more than 10 times lower. Section 2.10 shows that the expected results were a slow, gradual linear increase in memory consumption and not the explosive growth we see here.

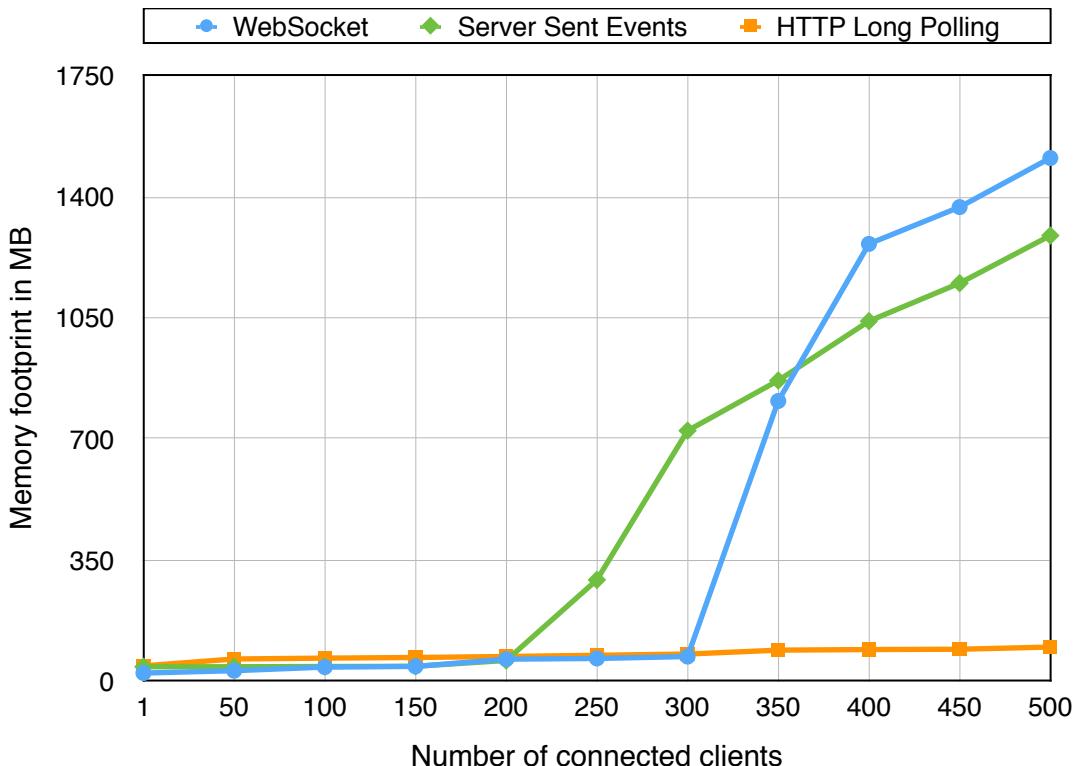


Figure 32: The memory footprint right after the first test scenario's test phase.

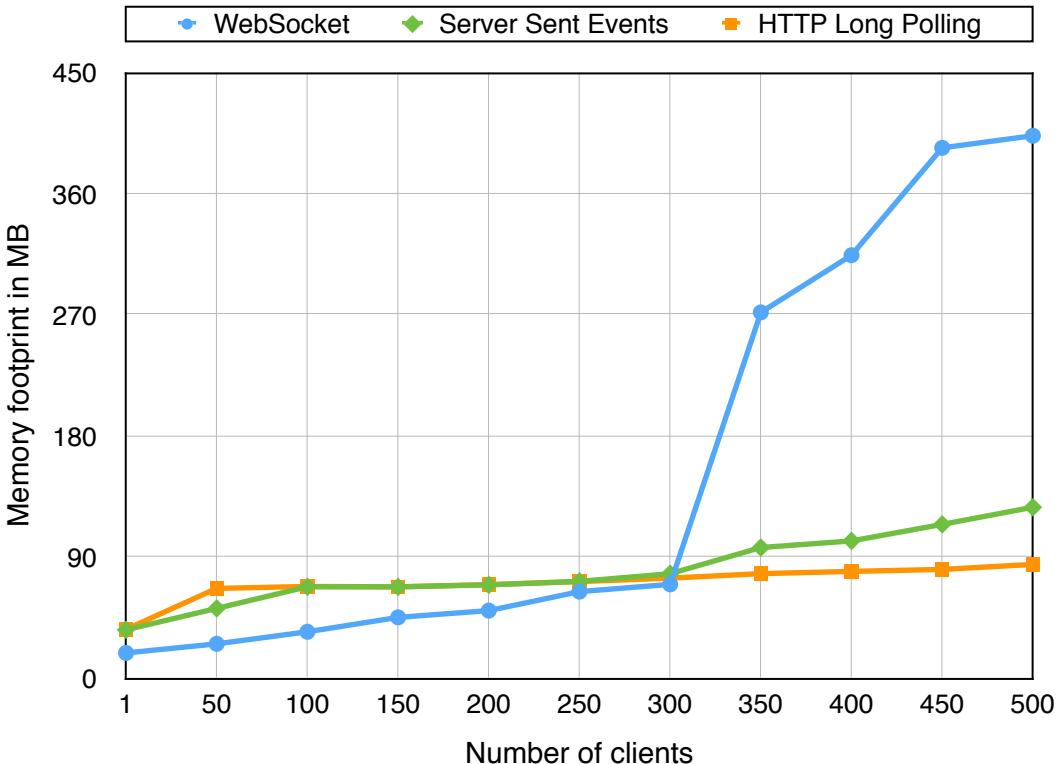


Figure 33: The memory footprint right after the second test scenario's test phase.

In the second test scenario, found in figure 41, we see the same story, although not in the same magnitude. When the Server Sent Events and WebSocket servers reach full CPU utilization at 350 clients, there is a bump in memory usage with the WebSocket version being 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 server is under stressing load levels, it is, once again, only the Long Polling server that behaves expectedly.

4.6 Result Summary

| | Load test | Stress test |
|-------------------|---|---|
| Scenario 1 | <ol style="list-style-type: none"> 1. WebSocket 2. Server Sent Events 3. HTTP Long Polling | <ol style="list-style-type: none"> 1. HTTP Long Polling 2. WebSocket 3. Server Sent Events |
| Scenario 2 | <ol style="list-style-type: none"> 1. WebSocket 2. Server Sent Events 3. HTTP Long Polling | <ol style="list-style-type: none"> 1. HTTP Long Polling 2. Server Sent Events 3. WebSocket |

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

The table above shows a ranking of the three transports 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 hungry than a Server Sent Events connection or a hanging Long Polling request, the difference is not that big.

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

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. At the end of the chapter, I will discuss how the two different scenarios were to implement from a programmer's perspective.

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 from section 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 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. And, they constantly used the longest time to answer the respond.

The two figures below show the response times from both test scenarios, but only where the tests were a load tests. This shows how far they could reach before the server was CPU limited.

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 stress load levels. As long as the levels of load are moderate, they all respond quickly.

In the second scenario, we see a similar story with the Long Polling server reaching 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.

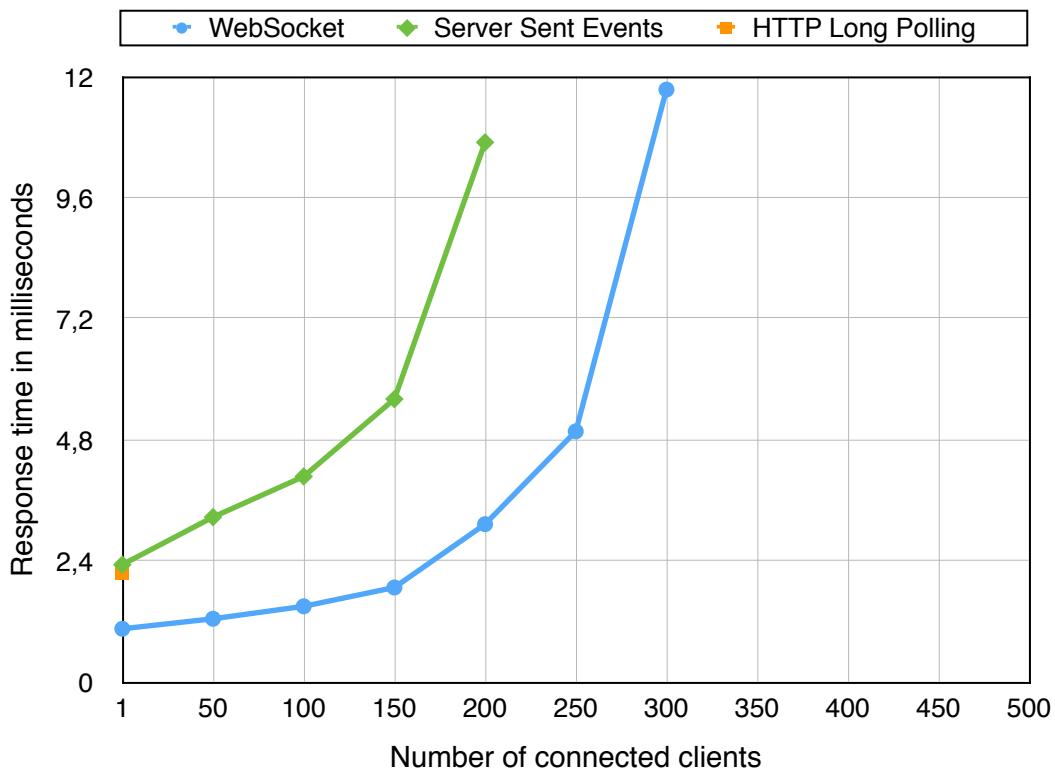


Figure 34: Scenario 1 load testing response time.

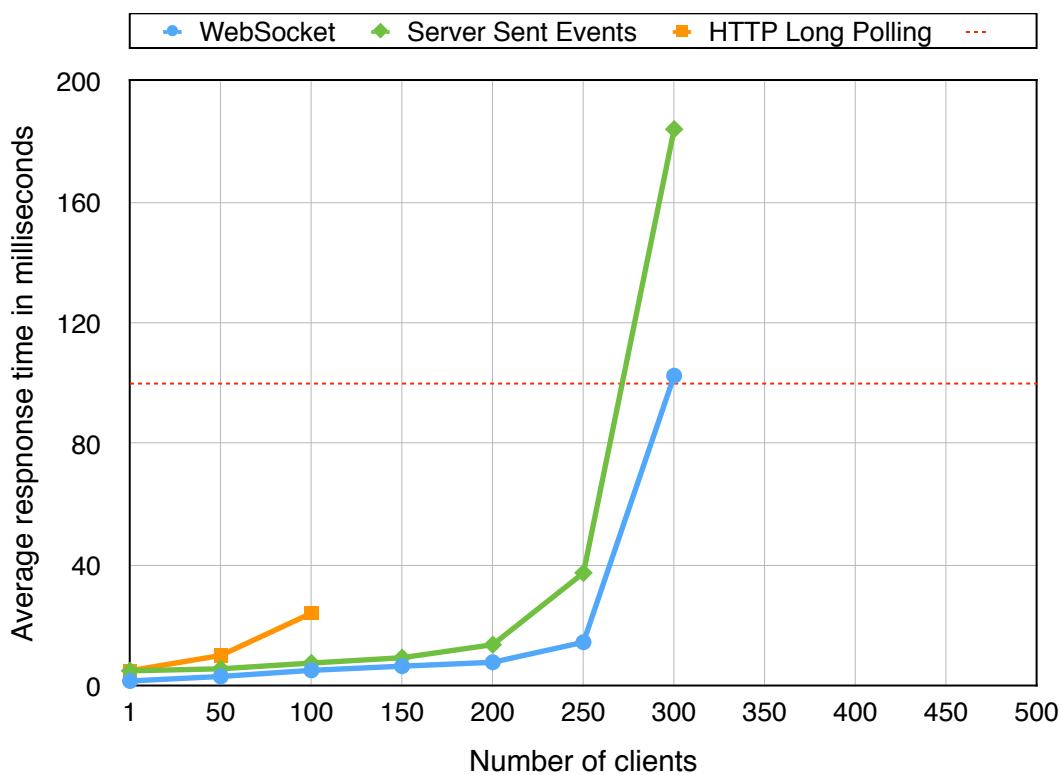


Figure 35: Scenario 2 load testing response time.

There are a few points to take away from these observations:

5.2.1 The Long Polling Server Performs Expectedly Bad

As expected (see section 2.10), the HTTP Long Polling server did perform poorly compared to the other two versions. This can be explained by the large amount of headers that needs processing, as well as incoming requests.

5.2.2 The Server Sent Events Server Performs Great

That Server Sent Events would perform close to WebSocket in the first scenario was expected, as the technology 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 did 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

Even though the WebSocket server introduces a small memory penalty compared to the other servers, the WebSocket servers were the most performant in the load tests. That was true, both in terms of CPU use and response time. These results were expected, as WebSocket is a protocol designed to be a thin layer on top of TCP. WebSocket seem to work great in a real-time setting.

5.3 Stress Testing

All the expectations from section 2.10 was 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. The figure in section 2.10 shows how the response time was expected to increase in a linear, but slow manner. In the stress testing, the Server Sent Events and WebSocket servers show a more dramatic increase in response time.

The two figures below show the response time in the first and second scenario during the test phase. But, I have removed the part where the servers are under moderate load, so they only show the results of the stress testing. In a sense they continue on from where the figures in section 5.2 left off.

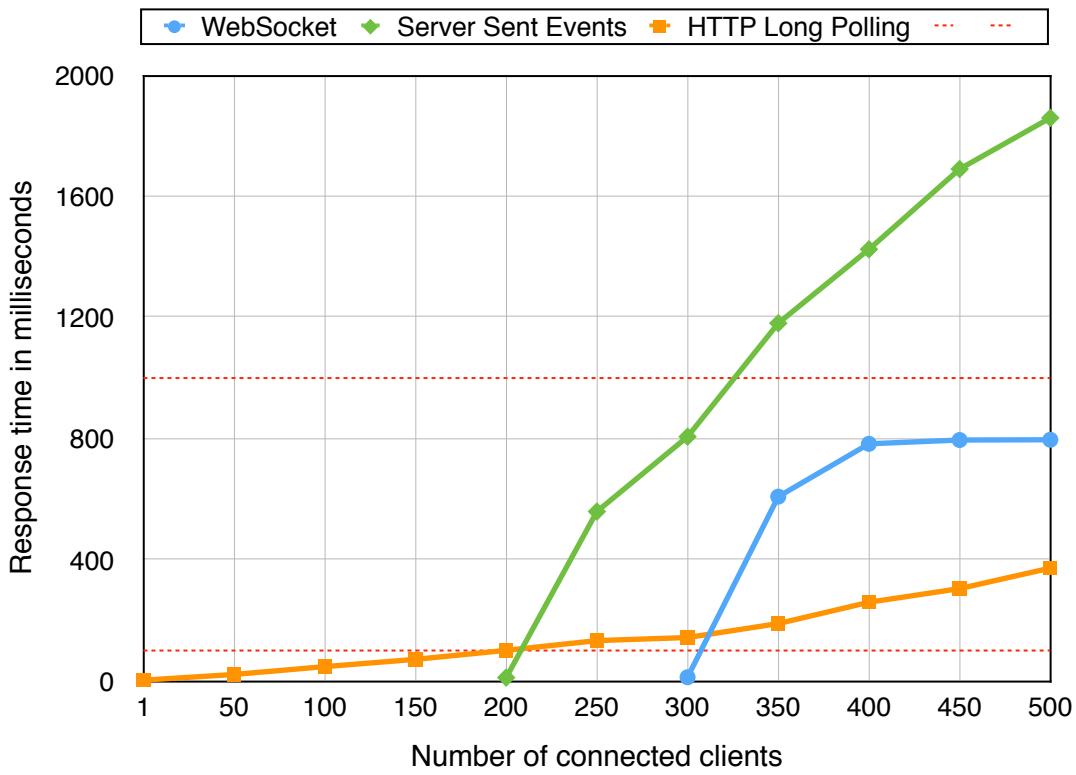


Figure 36: Scenario 1 stress testing response time.

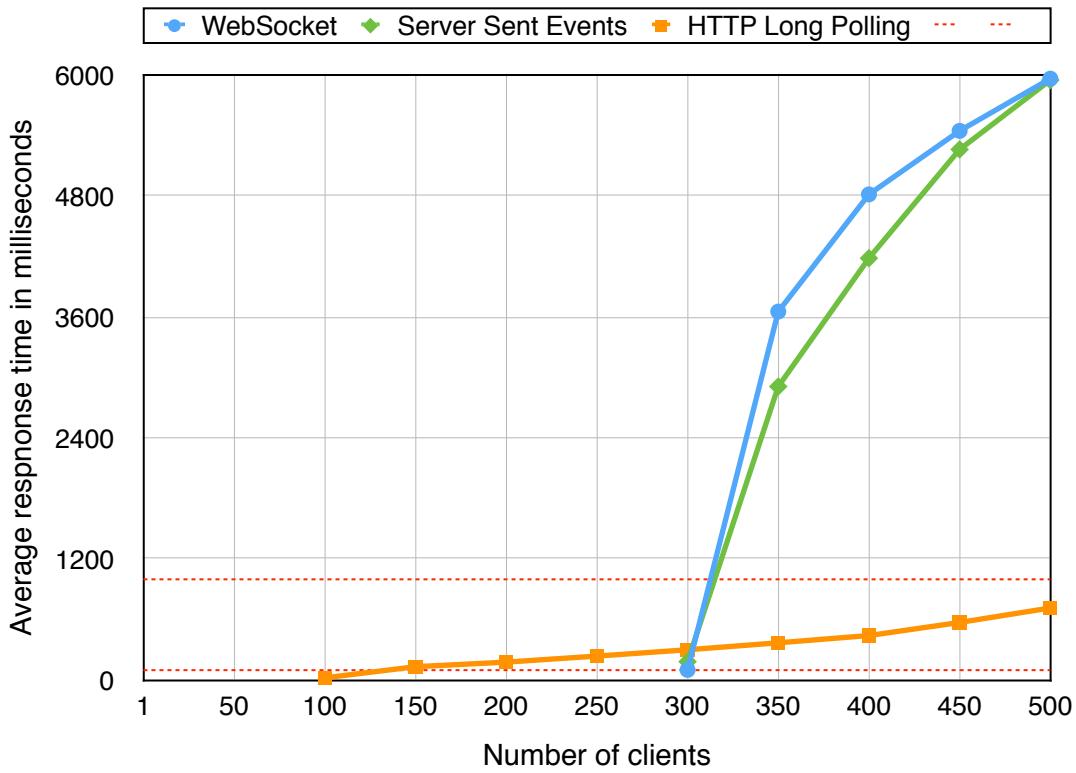


Figure 37: Scenario 2 stress testing response time.

5.3.1 Unexpected and Dramatic Increase in Response Time

The linear, slow and steady increase in response time the Long Polling server has, is the expected server behavior. The sudden spike in response times the other two servers demonstrate, was totally unexpected and very surprising. Following is the reasons why these results were unexpected:

The load tests show a totally different picture

The Server Sent Events and WebSocket servers performed very well in the load tests. In the stress test however, they perform really bad, quickly breaching the 0.1 and 1.0 second limits from section 2.9.2.

The HTTP Long Polling versions 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 have this problem, makes it very surprising.

5.3.2 Average vs. Median Response Time

Although the response time for the WebSocket and Server Sent Events servers grew steep in the first scenario, it was an even more dramatic growth in the second scenario. This 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 damaged 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 than the average, but they are both well above the numbers from the first 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 messages that are temporarily or permanently stored in memory. Temporarily for the Server Sent Events and WebSocket servers, and permanently for the Long Polling server. (See the example in figure 6 in subsection 2.4.1 to understand why).

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

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 explanations for why the response times suddenly goes through the roof.

The dramatic increase in memory footprint after the tests (see section 4.5) are happening at the same time 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 really no reason 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), makes me believe there is an error, a bug or another anomaly that compromises the stress test results.

5.4 Anomaly Discussion

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

5.4.1 Issue With the WebSocket Implementation

The WebSocket servers are the ones that are most affected by this anomaly, as the spikes in response time and memory occur in both test scenarios. This makes it possible that there is a bug or 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*”[45]. Memory leaks can cause the garbage collector to become more aggressive[46], meaning increased CPU use. If these issues did occur in my tests, they could possibly 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 3.7.1, I chose to implement the Server Sent Events server myself. Interestingly, this means that there are no difference in libraries used by the Server Sent Events server and the Long Polling twin (true for both test scenarios). Then arises the question of why the Server Sent Events version is affected and not the Long Polling twin.

Maybe there is 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[47] 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. Soon after, Joyent released Node.js version 0.12 with a more updated V8 engine. Maybe the engine running in io.js or the one in Node.js 0.12 fixes the test anomalies.

It is also possible that what was discussed in section 3.7.5 about the Event Loop blocking on the broadcast function, 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 with 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 with my own code. Writing bug free code is proven to be difficult, and especially when there are few people testing the programs. I do however not believe this is the case. 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 really hard, that the anomalies and unexpected results appear. This leads me to believe that, if there is a software error causing these anomalies, it is probably not in my code.

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

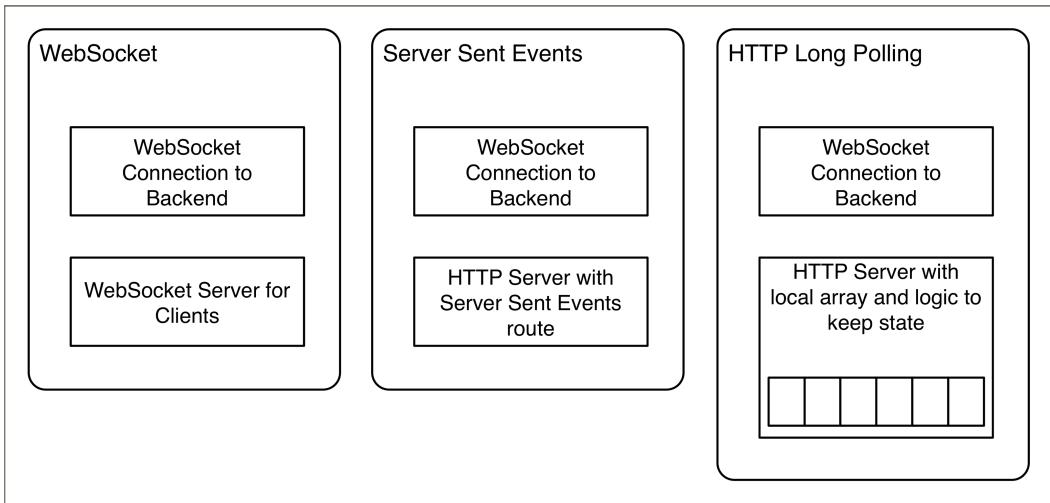


Figure 38: The three different servers in the first test scenario

The figure above 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, so that component is common among the three versions.

Both the Server Sent Events and WebSocket servers were straightforward to write. The two technologies both support the concept of a persistent connection, so there was no need to store incoming backend 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 for more than one reply after each request. This means double the network traffic and increased complexity on the server. Because of the issue presented by Figure 5 in section 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 server complexity, 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

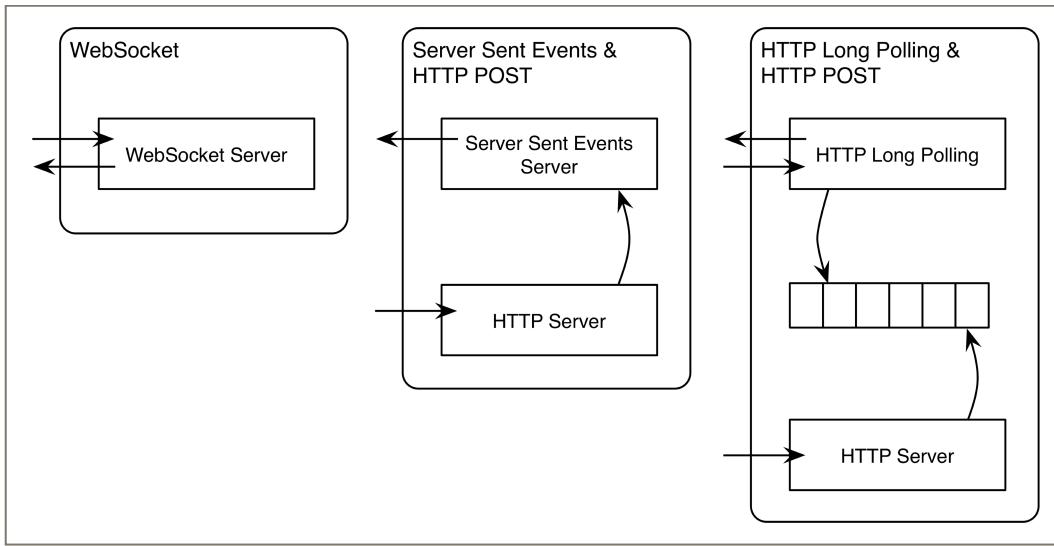


Figure 39: The three different servers in the second test scenario.

Looking at the figure above, the difference in server complexity between the three servers becomes really 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. Conceptually simple and easy to program.

Server Sent Events for outgoing and an additional HTTP POST route for incoming chat messages proved to be a great combo. Because Server Sent Events allow us to keep track of connections, it was easy to distribute chat messages as soon as they were received. 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 6 in subsection 2.4.3 proves, you need a local buffer where all messages are stored (at least temporarily) to ensure that every client gets them all. Conceptually more complex and more difficult to develop.

Chapter 6: Ending

6.1 Thesis Conclusion

In the thesis conclusion 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 chapter.

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

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

How does WebSocket perform compared to Long Polling and Server Sent Events in an unidirectional messaging setting with high levels of load?

When the load level is high, but not extreme, WebSocket as a transport performs better than both Server Sent Events and Long Polling, both in terms of CPU usage and response time. Server Sent Events is not too far behind, but Long Polling is.

When the load levels are at a maximum, WebSocket performs poorly compared to Long Polling, but comparable to Server Sent Events in my test scenarios. This is true for both response time and memory consumption.

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

When the load level is high, but not extreme, WebSocket as a transport, performs much better than Long Polling in terms of CPU usage and response time. Server Sent Events, on the other hand, performs very comparable to WebSocket, both in terms of CPU utilization and response time.

When the load levels are at a maximum, WebSocket performs poorly compared to Long Polling, but comparable to Server Sent Events in my test scenarios. This is true for both response time and memory consumption.

Does WebSocket provide any advantages over Long Polling and Server Sent Events from a programmer's perspective?

There are several advantages that WebSocket provide over Long Polling. First, the concept of a persistent connection. The connection makes WebSocket a stateful protocol. This is especially advantageous in a real-time application where the server would like to keep information about a client. The second advantage over Long Polling is the fact that

WebSocket is bidirectional by nature. This makes server push a feature of the protocol and not a technique, like Long Polling is. Third, WebSocket is a protocol with a well defined API, while Long Polling is not.

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

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 a maximum, WebSocket provides both performance and programmer friendliness benefits over Long Polling and Server Sent Events in all types of real-time applications.

However, if the CPU load levels reach a maximum, the performance penalty WebSocket sees in this thesis' scenarios, makes it a terrible choice in terms of performance, both from a user's perspective with long response times and from a server's perspective with very high memory footprint.

6.2 Further Work (in discussion?)

HTTP 2.0

WebRTC

Samme oppgave på flere software platformer

GC

1 til 2 sider.

Bibliography

1. Adam Bergkvist, D.C.B., Cullen Jennings, Anant Narayanan. WebRTC 1.0: Real-time Communication Between Browsers. 2014; Available from: <http://dev.w3.org/2011/webrtc/editor/archives/20140321/webrtc.html>.
2. Johannessen, K., Real Time Web Applications - Comparing frameworks and transport mechanisms, in Department of Informatics. 2014, University of Oslo.
3. Engin Bozdag, A.M.a.A.v.D., A Comparison of Push and Pull Techniques for AJAX. 2007.
4. Jõhvik, M., Push-based versus pull-based data transfer in AJAX applications. 2011.
5. Wikipedia: Internet protocol suite. 2015 [20.04.15]; Available from: http://en.wikipedia.org/wiki/Internet_protocol_suite.
6. HTML5. 2014; Available from: <http://en.wikipedia.org/wiki/HTML5>.
7. Berners-Lee, T. The HTTP Protocol As Implemented In W3. 1991; Available from: <http://www.w3.org/Protocols/HTTP/AsImplemented.html>.
8. Ltd., T. HTTP Scoop. 2014; Available from: <http://www.tuffcode.com>.
9. Garrett, J.J., Ajax: A New Approach to Web Applications. 2005.
10. W3C. XMLHttpRequest Level 1 - W3C Working Draft. 2014; Available from: <http://www.w3.org/TR/XMLHttpRequest/>.
11. T. Bray, E., The JavaScript Object Notation (JSON) Data Interchange Format.
12. Russell, A. Comet: Low Latency Data for the Browser. 2006 [20.04.15]; Available from: <http://infrequently.org/2006/03/comet-low-latency-data-for-the-browser/>.
13. W3C. Open Web Platform Milestone Achieved with HTML5 Recommendation. 2014; Available from: <http://www.w3.org/2014/10/html5-rec.html.en>.
14. W3C. Server-Sent Events. 2009; Available from: <http://www.w3.org/TR/2009/WD-eventssource-20091029/>.
15. Grigorik, I., High Performance Browser Networking. 2013: O'Reilly.
16. Vanessa Wang, F.S., Peter Moskovits, The Definitive Guide to HTML5 WebSocket. 2013: Apress.
17. I. Fette, A.M. RFC 6455. 2011; Available from: <https://tools.ietf.org/html/rfc6455>.
18. WebSocket protocol handshake. 2014; Available from: http://en.wikipedia.org/wiki/WebSocket#WebSocket_protocol_handshake.

19. IANA. WebSocket Protocol Registries. 2014; Available from: <https://www.iana.org/assignments/websocket/websocket.xml>.
20. Dart. Dart: Structured web apps. 2015 [09.04.2015]; Available from: <https://www.dartlang.org>.
21. CoffeeScript. CoffeeScript. 2015 [09.04.2015]; Available from: <http://coffeescript.org>.
22. Microsoft. Welcome to TypeScript. 2015 [09.04.2015]; Available from: <http://www.typescriptlang.org>.
23. Dahl, R. Node.js. in JSConf.eu. 2009.
24. Google. V8 JavaScript Engine. Available from: <https://code.google.com/p/v8/>.
25. PayPal. Node.js at PayPal. 2013 [13.04.15]; Available from: <https://wwwpaypal-engineering.com/2013/11/22/node-js-at-paypal/>.
26. Google. Chrome V8 Design Elements. 2012 [15.04.15]; Available from: <https://developers.google.com/v8/design>.
27. J. D. Meier, C.F., Prashant Bansode, Scott Barber, Dennis Rea, Performance Testing Guidance for Web Applications. 2007.
28. Nielsen, J., Response Times: The 3 Important Limits. 1993.
29. Sharp, R. nodemon. 2015 [13.04.15]; Available from: <http://nodemon.io>.
30. GitHub. Most starred repositories. 2015 [13.04.15]; Available from: <https://github.com/search?utf8=%E2%9C%93&q=+stars%3A%3E1000>.
31. npm. Node Package Manager. 2015; Available from: <https://www.npmjs.com>.
32. Apple. JavaScript for Automation Release Notes. 2014 [13.04.15]; Available from: <https://developer.apple.com/library/mac/releasenotes/InterapplicationCommunication/RN-JavaScriptForAutomation/>.
33. Crockford, D., JavaScript: The Good Parts. 2008: O'Reilly.
34. Joyent. Readline Node.js. 2015 [13.04.15]; Available from: <https://nodejs.org/api/readline.html>.
35. Texin, T. Unicode Supplementary Characters Test Data. 2010 [cited 2015 12th january 2015]; Available from: <http://www.i18nguy.com/unicode/supplementary-test.html>.
36. Qveflander, N., Pushing real time data using HTML5 Web Sockets. 2010.
37. Stangvik, E.O. ws: a node.js websocket implementation. 2014 [23.03.15].
38. Stangvik, E.O. websocket client benchmark. 2015; Available from: <http://einoras.github.io/ws/benchmarks.html>.

39. Hellesøy, A. EventSource-node. 2015 [13.04.15]; Available from: <https://github.com/aslakhellesoy/eventsource-node>.
40. StrongLoop. Express - Node.js web application framework. 2015 [23.03.15]; Available from: <http://expressjs.com>.
41. Request - Simplified HTTP client. 2015 [13.04.15]; Available from: <https://github.com/request/request>.
42. Modulus. process-monitor. 2015 [13.04.15]; Available from: <https://github.com/onmodulus/process-monitor>.
43. Codenomicon. The Heartbleed Bug. 2014 [22.03.15]; Available from: <http://heartbleed.com>.
44. StrongLoop. Node.js Performance Tip of the Week: Event Loop Monitoring. 2014 [15.04.15]; Available from: <https://strongloop.com/strongblog/node-js-performance-event-loop-monitoring/>.
45. Kazemier, A. Changes in ws 0.5. 2014 [30.03.2015]; Available from: <https://github.com/websockets/ws/commit/d242d2b8ddaa32f7f8a9c61abe74615767a91db4#diff-e1bbd4f15e3b63427b4261e05b948ea8>.
46. Mozilla. Tracking Down Memory Leaks in Node.js – A Node.JS Holiday Season. 2012 [29.03.15]; Available from: <https://hacks.mozilla.org/2012/11/tracking-down-memory-leaks-in-node-js-a-node-js-holiday-season/>.
47. io.js. io.js - JavaScript I/O. 2014; Available from: <https://iojs.org/>.

Appendix

List of Acronyms

| | |
|-------------|-----------------------------------|
| AJAX / Ajax | Asynchronous JavaScript and XML |
| DOM | Document Object Model |
| HTTP | Hypertext Transfer Protocol |
| JSON | JavaScript Object Notation |
| SSE | Server Sent Events |
| TCP | Transmission Control Protocol |
| W3C | World Wide Web Consortium |
| WS | WebSocket |
| XML | Extensible Markup Language |
| IO | Input/Output |
| API | Application Programming Interface |
| JVM | Java Virtual Machine |

Code

All the test code, as well as digital versions of the thesis is available for download at GitHub.
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. It might also be required to increase the OS limit for user processes.

3.9.1 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
```

3.9.2 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 |