Real-Time Web with WebSocket

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

Contents

Abstract

Text

List of Figures

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 messages from the clients and as soon as one 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 distributed to the client *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 messages. These two examples show the types of real-time applications this thesis focuses on.

The web 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 that 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, but Long Polling is the near real-time HTTP technique I have chosen to use in my methodology. This is because Server Sent Events really is HTTP Streaming behind a nice API. 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, 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.

Which one of these transports can handle the highest number of clients? And which one of these transports respond quickest to a user request? These are a few questions related to performance. Further details are found in section 2.9 and chapter 3.

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

The first question is the main research question and the following 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 the Background chapter.

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 their possible reasons. This chapter also includes the thesis conclusion, where I directly answer the research questions. Lastly, there is a section on suggestions for 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 your 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 on 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 is 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 regards 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. In it 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 Jorvik 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 our thesis' similarity, I had contact with Johannessen 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 chose 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 of 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 and 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 following two lines are pseudocode (and a simplification) that shows the concept of how easy it is connect to a remote server and immediately send data over the socket:

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

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 a bunch of the apps on your own smartphone are powered by HTML, CSS and JavaScript as a web app 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. Along with HTML came HTTP, the protocol designed to retrieve HTML documents. HTTP works great for simple document fetching but is not designed for the advanced use cases of todays web apps. As this thesis will reveal, it is hard and suboptimal to develop real-time apps using HTTP.

HTML5 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 networking 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 request is marked with a status code to indicate whether the request was successful or not.

GET was the first HTTP method[6] 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
<blank line>
```

Figure 1: A GET request to www.uio.no. The blank line indicates end of message.

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 don't 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 resource use. But it can also be a problem, as we shall see.

Header fields are an important part of HTTP. The header fields are meta data that are added into the HTTP requests and responses. The «Host» line in figure 1 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 shows the entire GET request a web browser sends when going to www.uio.no. The web browser adds several header fields:

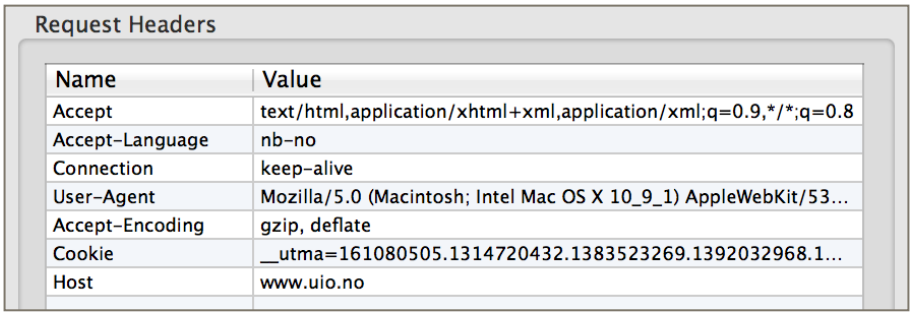

Figure 2: HTTP GET request to www.uio.no. Captured using HTTPScoop[7].

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:

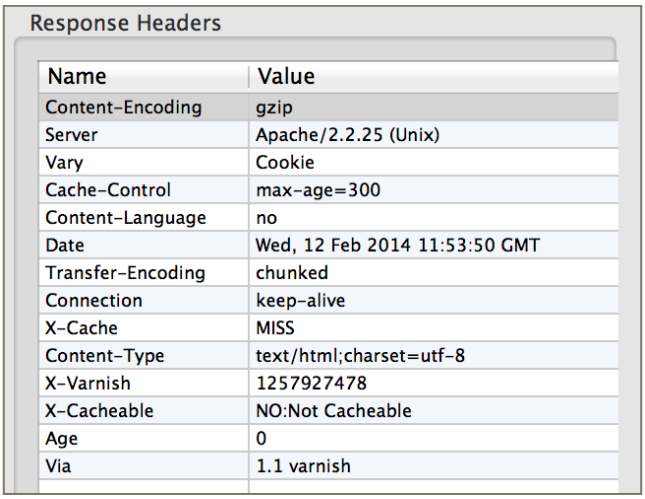

Figure 3: HTTP GET response from www.uio.no. Captured using HTTPScoop.[7]

2.4 Modern Web

As mentioned, HTTP was designed to serve static hyperlinked documents. Today however, 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 real-time behavior - Ajax.

2.4.1 Ajax

Ajax (Asynchronous JavaScript and XML) as a term was introduced by Jesse James Garret in 2004[8]. 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[9]. 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 HTML view (DOM) with new data. Even though Ajax has XML in its name, the

type of data are not limited to just XML; today JSON[10] 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 app 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 apps, 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 app 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 just set up a full duplex TCP socket and listen for updates. Even though HTTP utilizes TCP on the transport layer, HTTP itself is just half-duplex and there is no way for the server to push messages to the client. All server-sent messages must be a response to a client sent request. So how come, web apps like Twitter and Facebook seems 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 a client side JavaScript timer that periodically starts a function that polls 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 don't know when the one you're 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 messages.

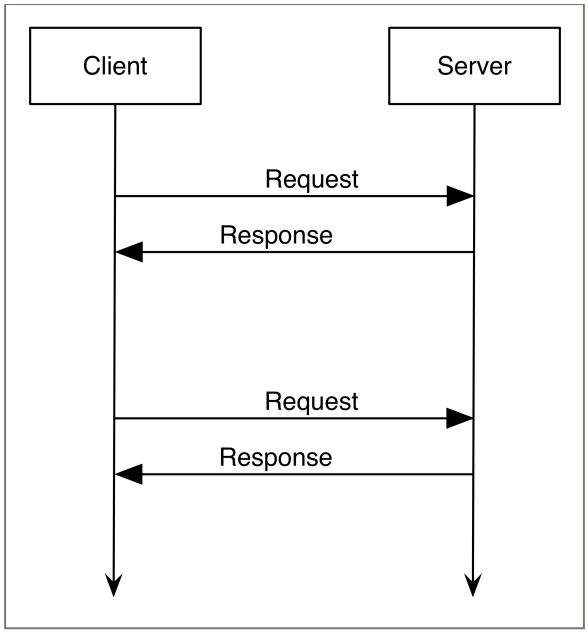

Figure 4: HTTP Polling example

2.5.2 Comet

When you need real time server push in you 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[11] 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 response is received, the client sends a new server 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.

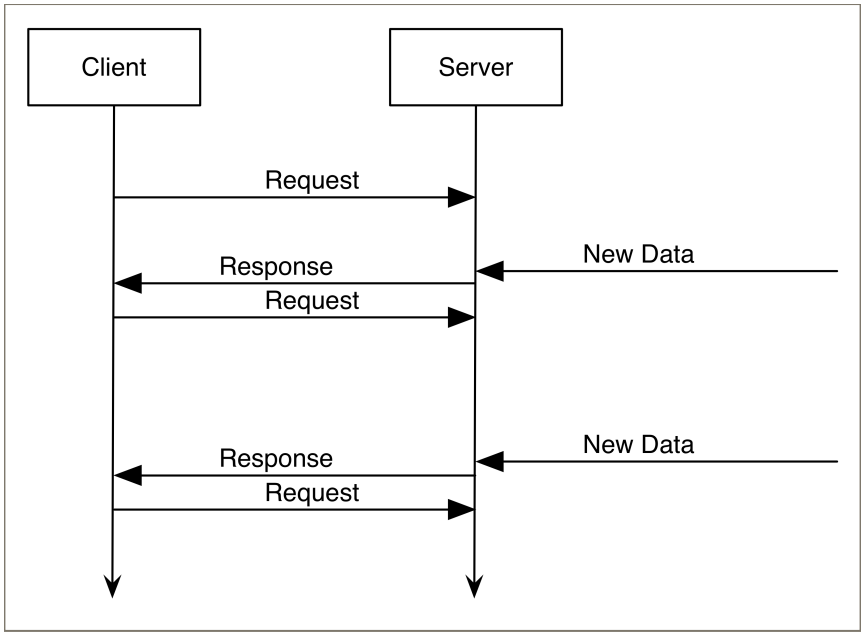

Figure 5: HTTP Long Polling Example

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, what is the problem? 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.

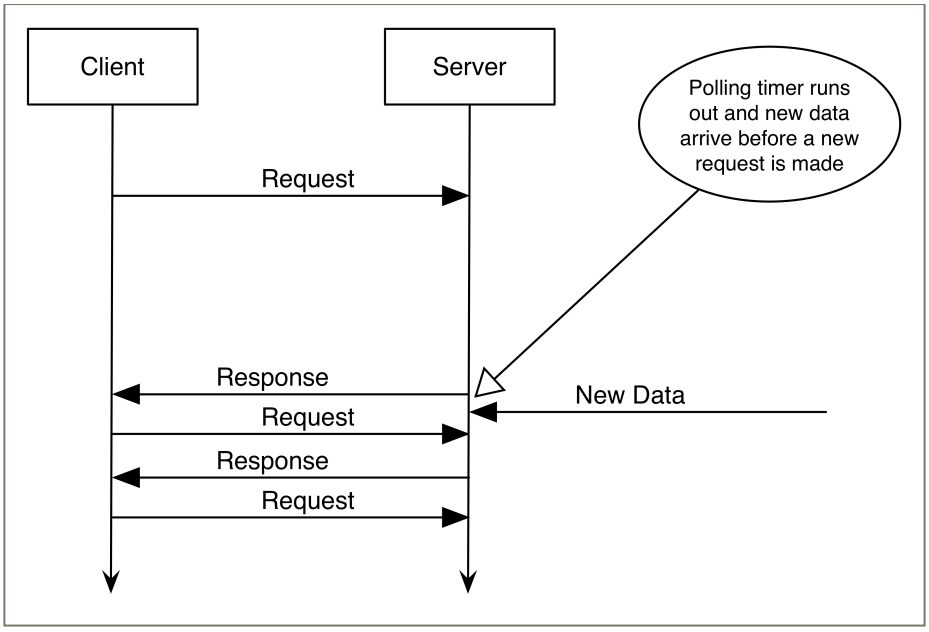

Figure 6: Example of client side outdated data with Long Polling

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 cram 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.

Even though Long Polling and HTTP Streaming accomplishes push behavior, 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 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[12]. Even though HTML5 adoption stated 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 is intended to clean up the syntax and unify web technologies as well as introduce 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 supports only server-to-client messaging. To understand Server Sent Events, you must understand the EventSource API 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 7: The EventSource API[14].

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 built for server-to-client messaging only, so there are 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
}
```

Figure 8: How to connect to a SSE endpoint and listen for updates.

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 figure.

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"
```

Figure 9: Part of example found in High Performance Browser Networking[15].

In the example above you can see how the server sets the client reconnect interval to 15 seconds. The examples also shows that data 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 (IE) supports Server Sent Events, but since IE 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 communications support. WebSocket utilizes a single TCP socket and simplifies some rough edges of the underlying protocol, with a simple, powerful, elegant API. WebSocket seems to be the protocol that eradicates the advantage platform native applications had compared to web applications. It promise 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's 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 10: 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. 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
}
```

Figure 11: How to open a WebSocket connection to example.com and set up an open event trigger.

2.7.2 The WebSocket Protocol

With the API explained, it's time to look into the protocol itself. 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's called the WebSocket Opening Handshake.

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 what's called the WebSocket opening handshake. Actually, this was a simplification since there's also an exchange of keys going on. This 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: HSmrc0sMlYUkAGmm5OPpG2HaGWk=
```

Figure 12: WebSocket opening handshake example[18]

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:

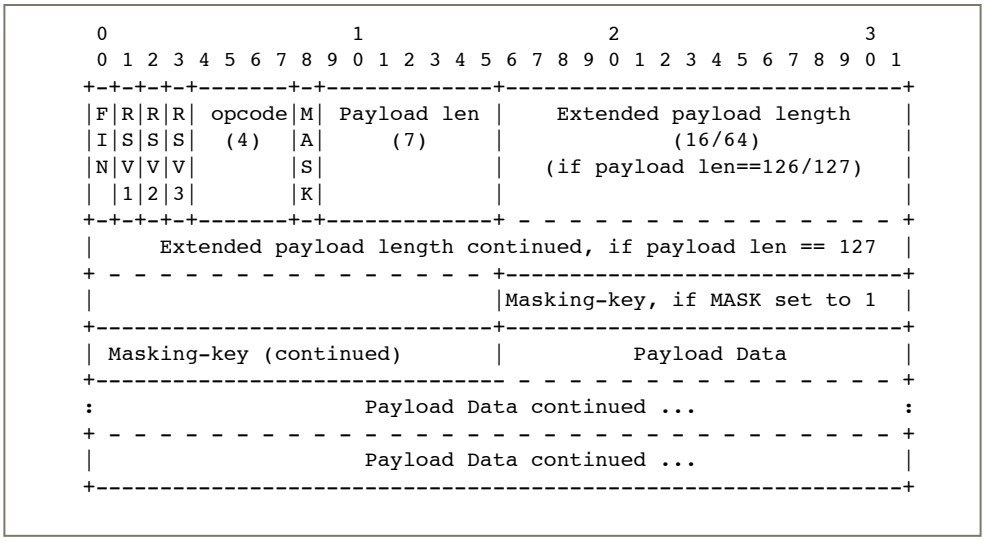

Figure 13: The WebSocket frame as defined in RFC6455[17]

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']);
```

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.

Together with subprotocols, there's 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
```

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 a 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”
}
```

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”}
```

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
```

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. 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”}
```

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. 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 Node.js

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 don’t understand these languages, so they ultimately need to be compiled to JavaScript. Same goes for HTML and CSS. JavaScript is in a sense the assembly language for 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.

As web applications became more and more complex following Web 2.0, 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 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++[23], 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.

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;
```

Figure 14: Blocking code

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

Figure 15: 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. Front end 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[24]. 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.

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"[25]. 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.

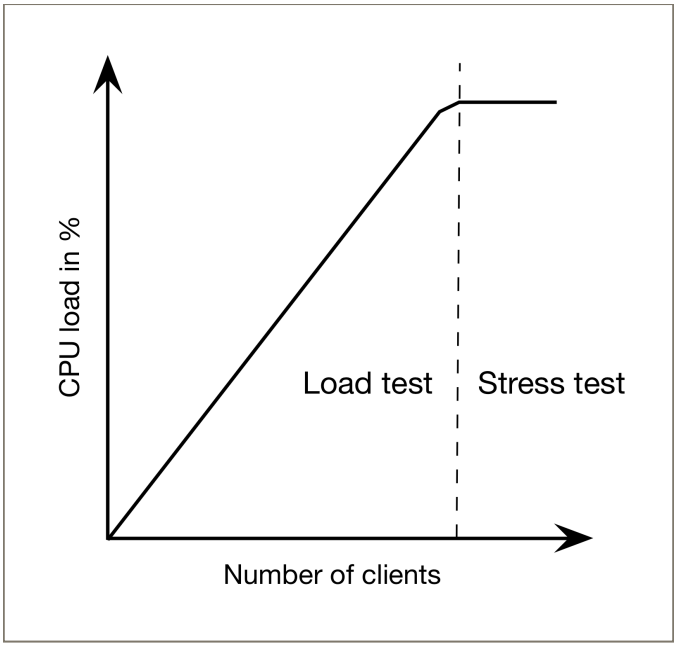

Figure 16: Relationship between load and stress tests

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[26] 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

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 low as well. I expect Server Sent Events to perform well, but not quite on level with WebSocket. This is because WebSocket 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 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 test 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 I have created. 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 the 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 individual implementation details, 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.

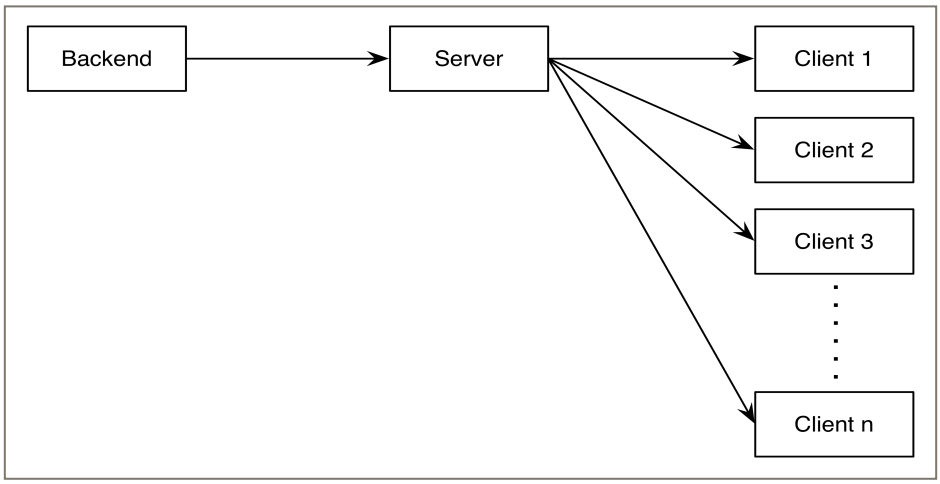

Figure 17: Simple diagram showing the three components. Messages from the backend are broadcasted to all clients.

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 individual implementation details, 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.

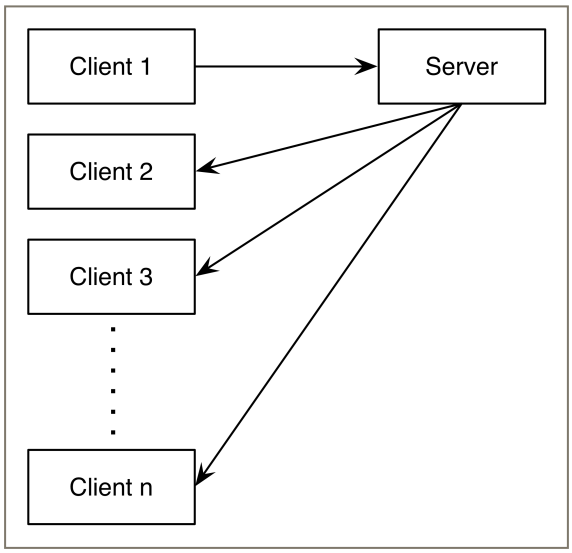

Figure 18: Simple diagram of the two components involved in the chat scenario. Client 1 sends a chat message to the server and the server broadcasts this to the other connected clients.

3.2 Test Data

The 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 with these 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 on an action.

To summarize, there are three data points that are collected, the CPU load and memory footprint on the server side, and the response time from the client side.

3.2.2 Collection Through Three Test Phases

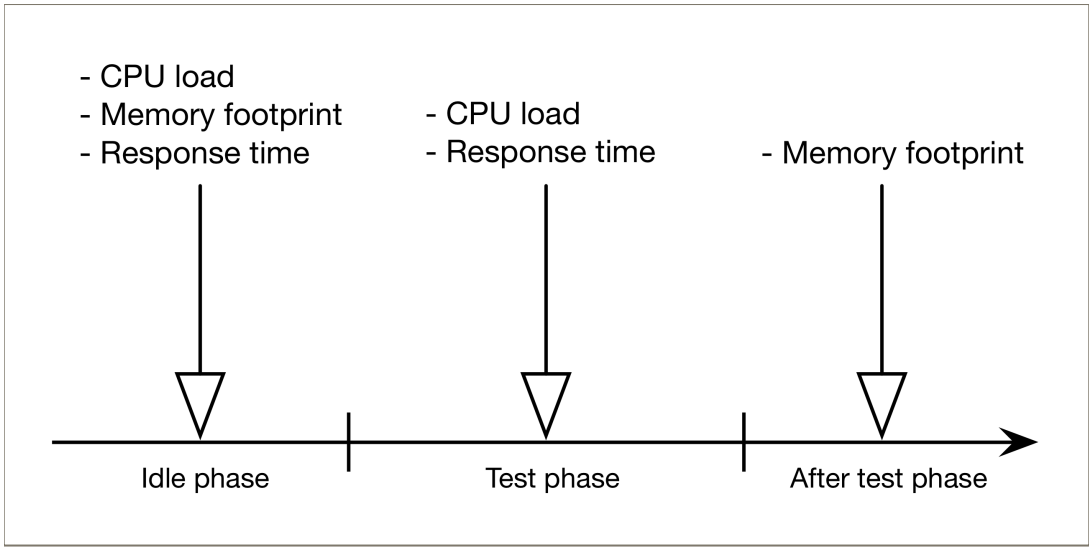

Figure 19: The three test phases and what is measured in each phase.

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 CPU load and response time is 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 be higher. 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 calculating 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, response time, is collected differently in the two test scenarios. In the first scenario, there is a separate ping client that every 50th millisecond sends a 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 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 both sounds 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 software platform. This subsection discusses why Node.js is a good match for this thesis.

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

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

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

It is also worth mentioning that JavaScript is an interpreted language. This can make development fast, especially with tools like nodemon[28].

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 said 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[29], 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[30], 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[31]. Also, I chose it so that I can learn some of its quirks[32], 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[33]. 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

With the tests being load and stress tests, they had to be run in such a way that it was possible to reach a server resource limit and a break point in response time with the chosen hardware. In this section, I will present and discuss the parameters for both test scenarios.

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 allows 709 user processes and on the client machine, about 220 user processes was constantly running. To then spawn 500 client processes was not allowed 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
```

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 have the maximum number of clients to be 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 three 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 In Between Messages

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

The Size of Each Broadcast 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[34]. 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

To make the test run long enough to minimize inaccuracies in CPU use caused by the garbage collector, but at the same time make it feasible for the time I had at hand, I set this number to 5000. With 5 milliseconds between each message, this means that each test run 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 landed on 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.

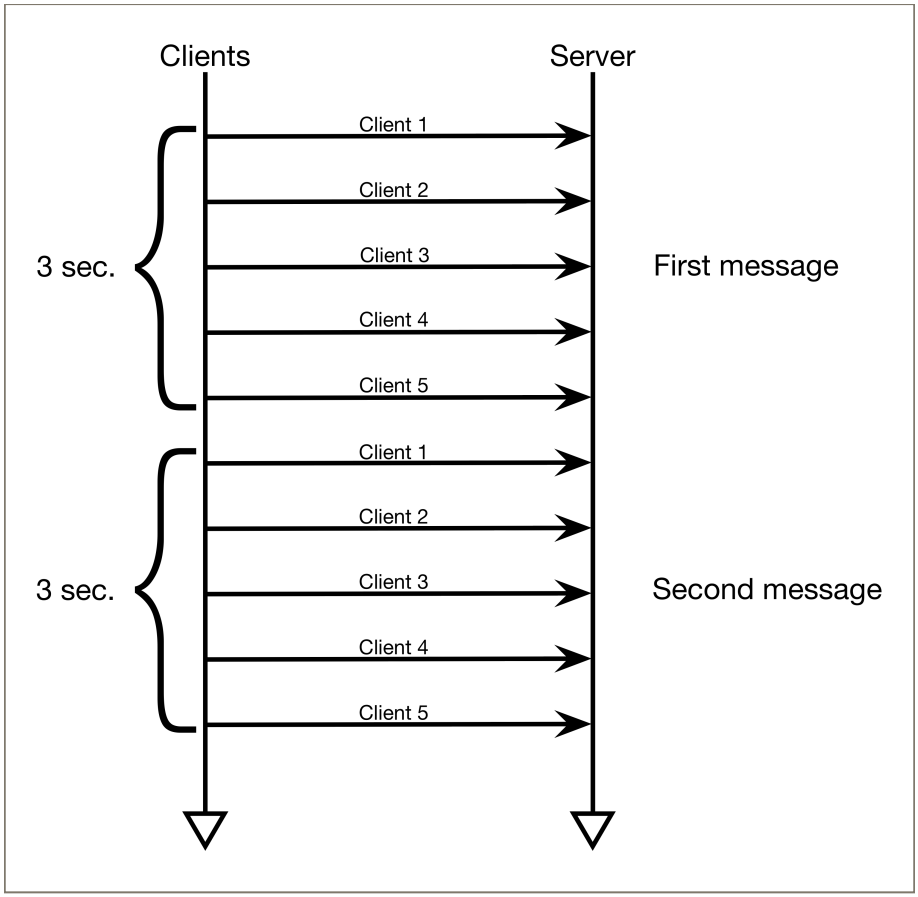

Figure 20: Example showing five clients sending their first two messages.

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.
- 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.

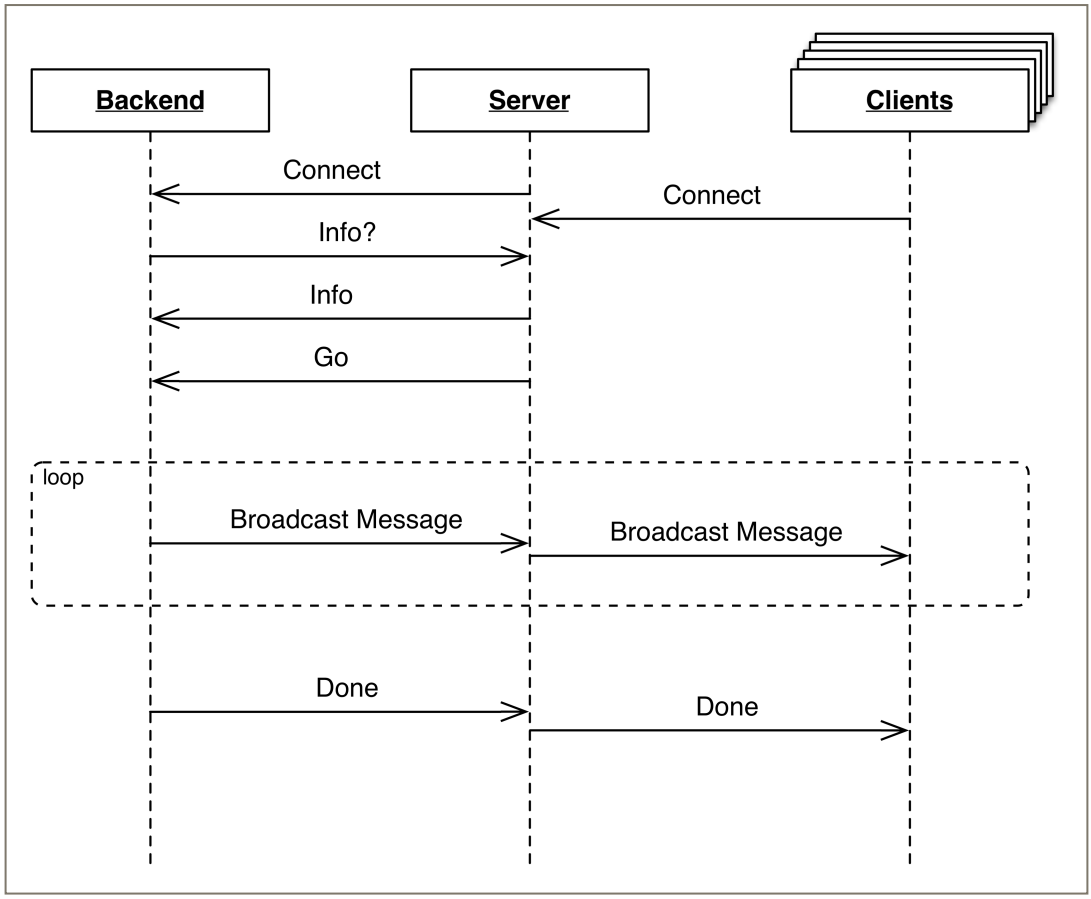

Figure 21: Sequence diagram showing detailed information flow. For simplicity the master client is not included.

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:

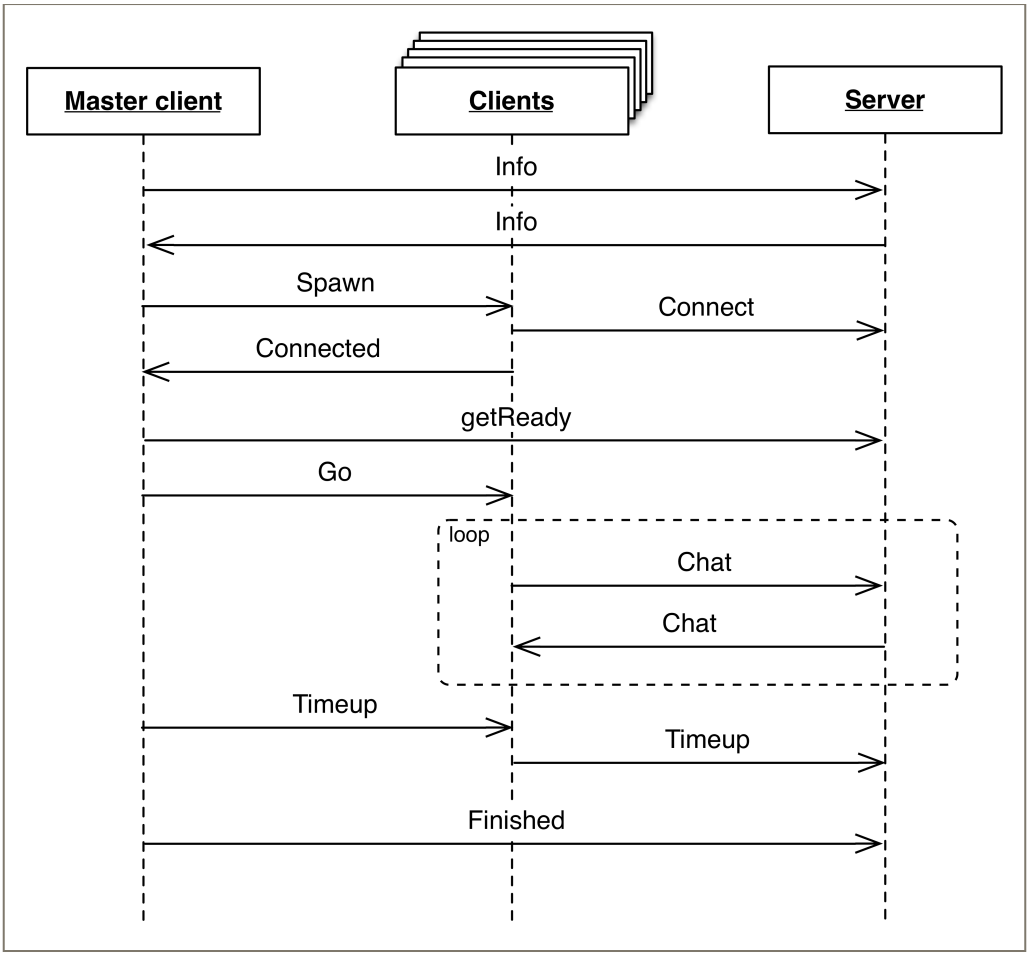

Figure 22: Sequence diagram showing the major flows of information for the chat test.

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 *timeup* 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 brings up topics with regards to 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. There is also a lot of common code and libraries used between the two tests:

3.6.1 Common Between Scenarios

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 Kristian Johannessen did.

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[35]. Thankfully Node.js has a large and dedicated community, so finding WebSocket libraries was easy. Socket.IO is already mentioned, but it 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[36] 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[37] 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[38]. The library is small and doesn't add anything on top of the protocol 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');
});
```

Figure 23: From sseserver.js - the Server Sent Events Endpoint

HTTP

There was a need for several routes into the server, and the popular web framework *Express[39]* helped to make that a simple reality. In addition, the small library *request[40]* made it easy to quickly send HTTP POST and GET requests.

Resource monitoring

To monitor resource usage on the server, the Node.js package Process Monitor[41] was used. It provides a simple interface to get CPU and memory usage of a process, using the UNIX application 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

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:

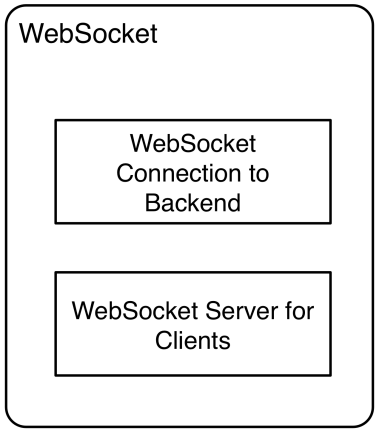

Figure 24: The WebSocket server in scenario 1

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:

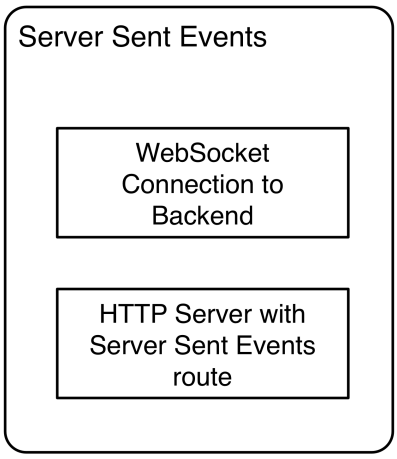

Figure 25: The Server Sent Events server in scenario 1

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:

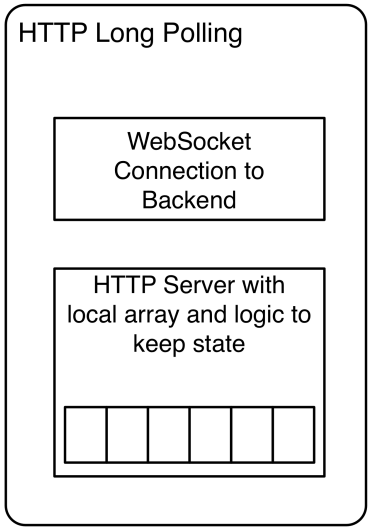

Figure 26: The HTTP Long Polling server in scenario 1

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

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.

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 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.

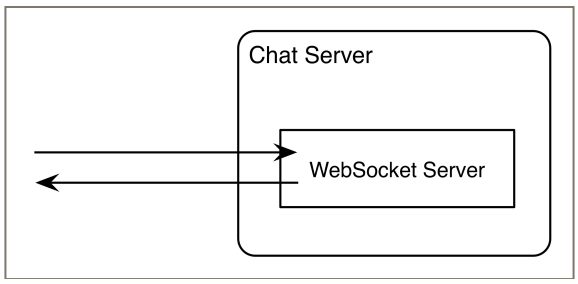

Figure 27: The WebSocket server in scenario 2

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.

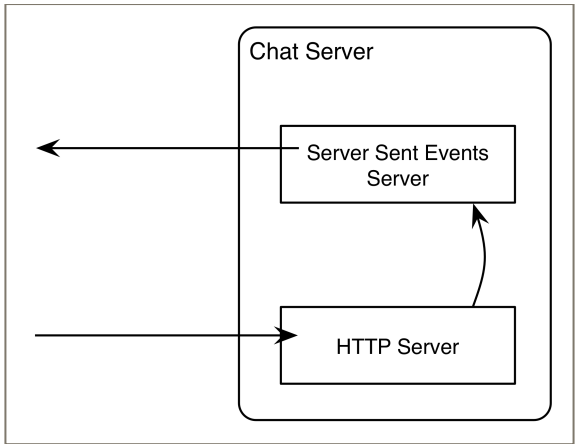

Figure 28: The Server Sent Events server in scenario 2

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.

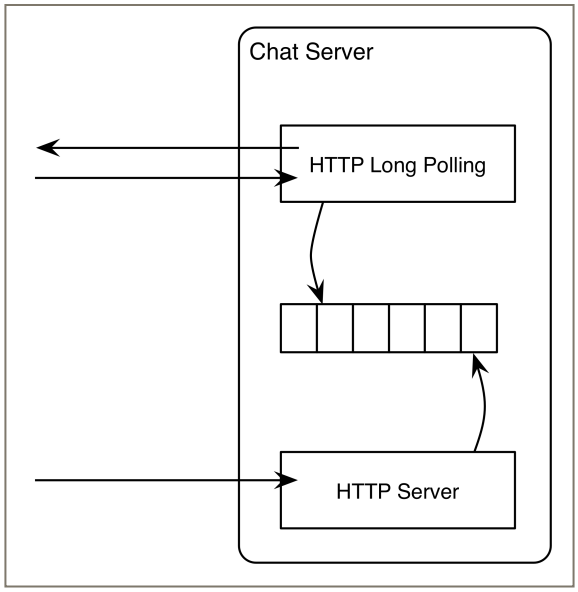

Figure 29: The HTTP Long Polling server in scenario 2

3.7 Limitations

Because of time constraints and the choices I have made to finish this thesis in time, there are some limitations to the methodology described 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. I did however, not want to measure this, as it relies heavily on the transport's implementation. A small memory leak, for example, can damage the results severely. Also, the actual testing would take up a very long time, making it unsuitable for this master's thesis.

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 between the three transports. However, for simplicity's sake the tests was 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[42]. 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 is a good 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.

A second limitation is the fact that I have chose to use just one software platform for my tests. This choice was influenced by a recommendation from Kristian Johannessen[2], but can be a limiting factor. Consider the scenario where the three tested transports' performance with Node.js is far off the average of all software platforms. In that case, the results in the next chapter would be an outlier and not indicate the real transport performance.

3.7.5 Node.js

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

First, the V8 JavaScript engine powering Node.js employs a garbage collector. This means that memory allocation and deallocation is handled by the runtime. V8 uses the stop-the-world collection scheme[43]. This means that V8 stops all program execution once the garbage collector runs. Because of this, response time may occasionally increase when the garbage collector does its work. CPU load may also be affected by the garbage collection.

The Event Loop is another 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.

Node.js is also very young. This means that tools are not as mature as they are on other server platforms.

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.

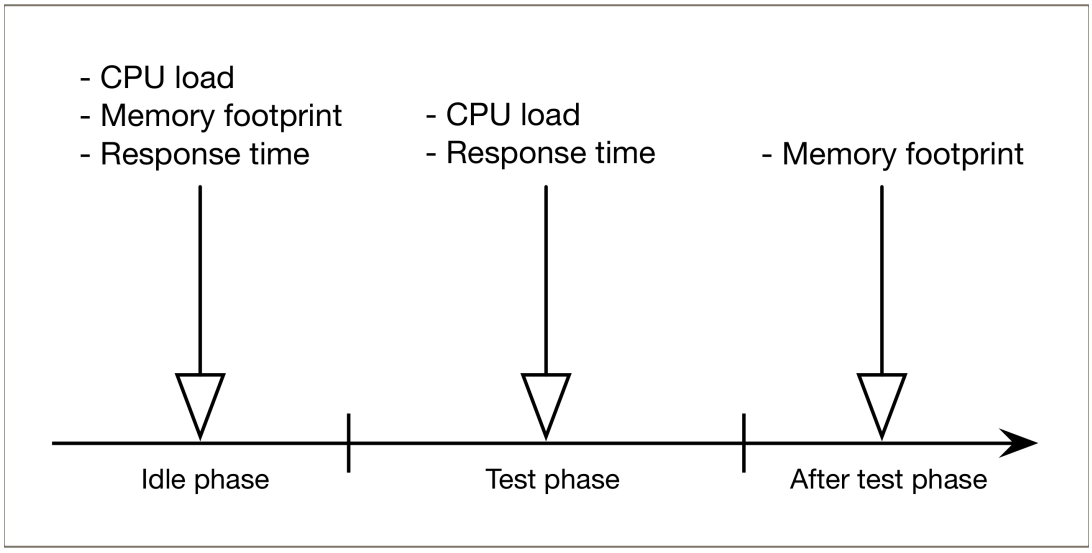

Figure 30: The three test phases and what is measured in each phase.

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

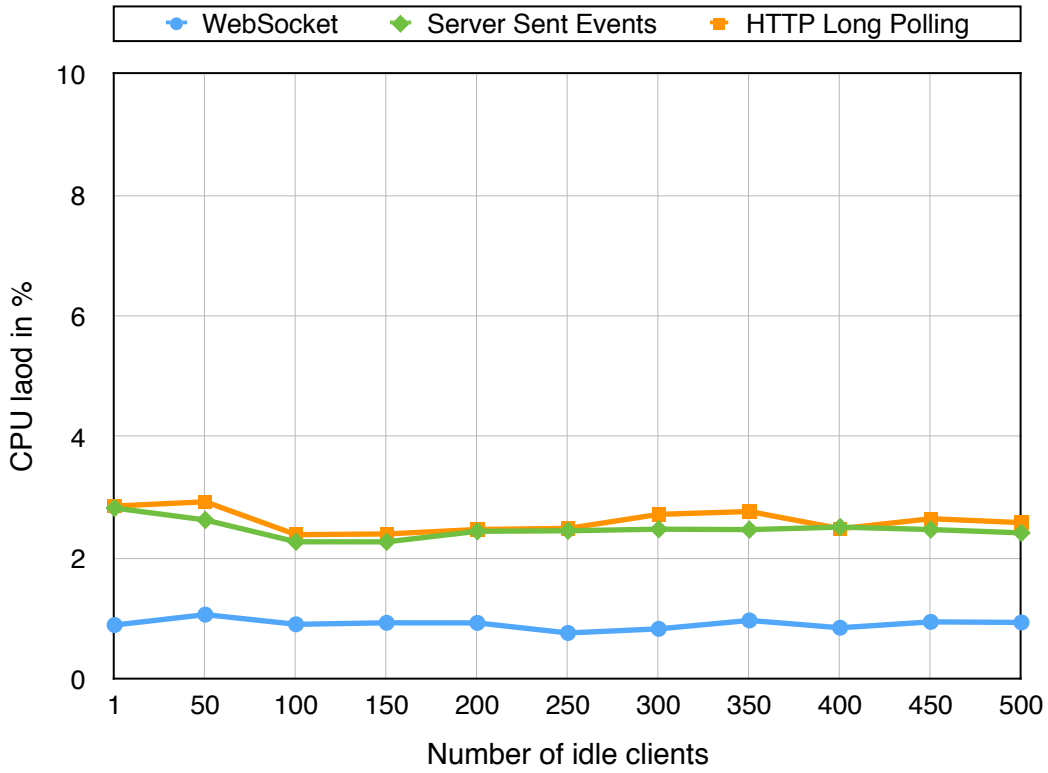

Figure 31: The CPU load for all three transports during the idle phase.

All three servers used so few CPU cycles that the Y-scale had to be set to a maximum of 10% to 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[39] 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. Once a client connects to a server and stays inactive, there is no work for the server CPU to do. The only are where I expected idle clients to have an effect, was in memory consumption.

4.2.2 Memory Footprint

Figure 32: 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

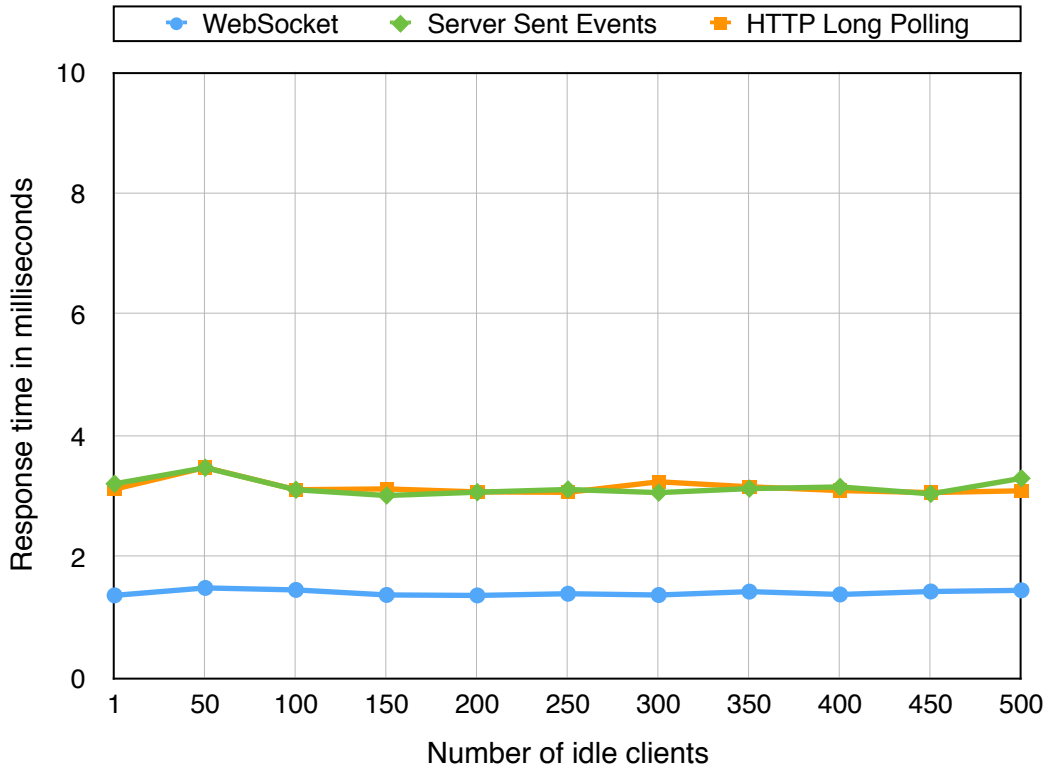

Figure 33: The response times for all three transports during the idle phase

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 was 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

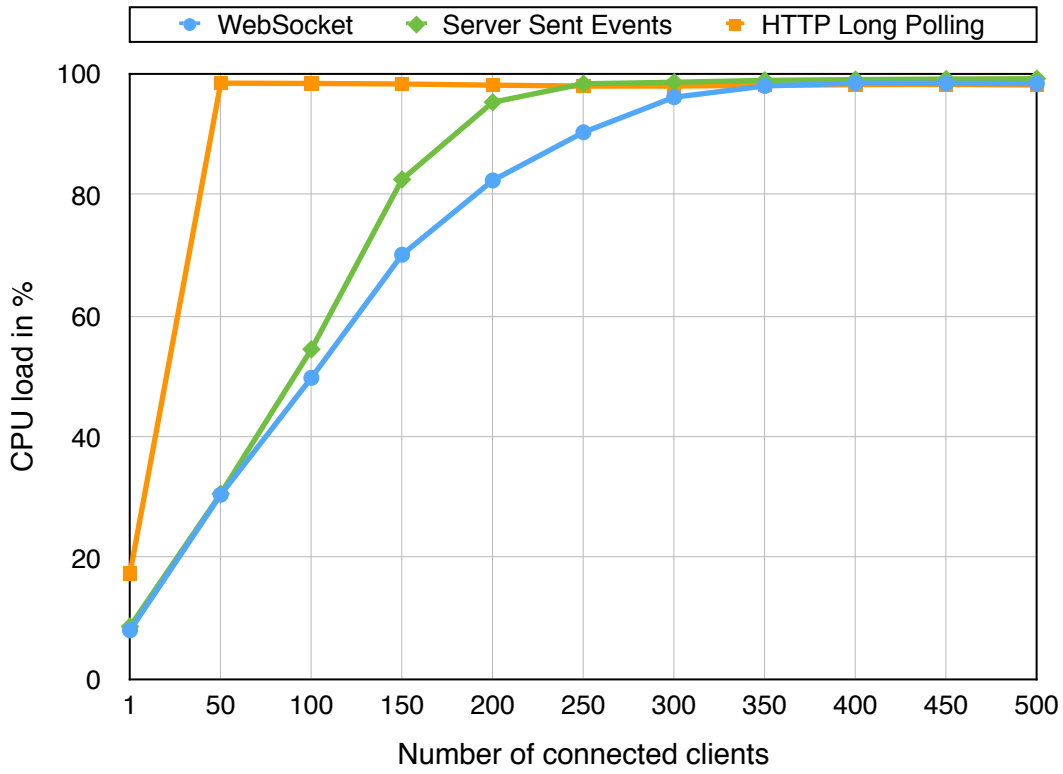

Figure 34: The CPU load during the first test scenario's test phase

The immediate feature of this chart to notice, is that 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

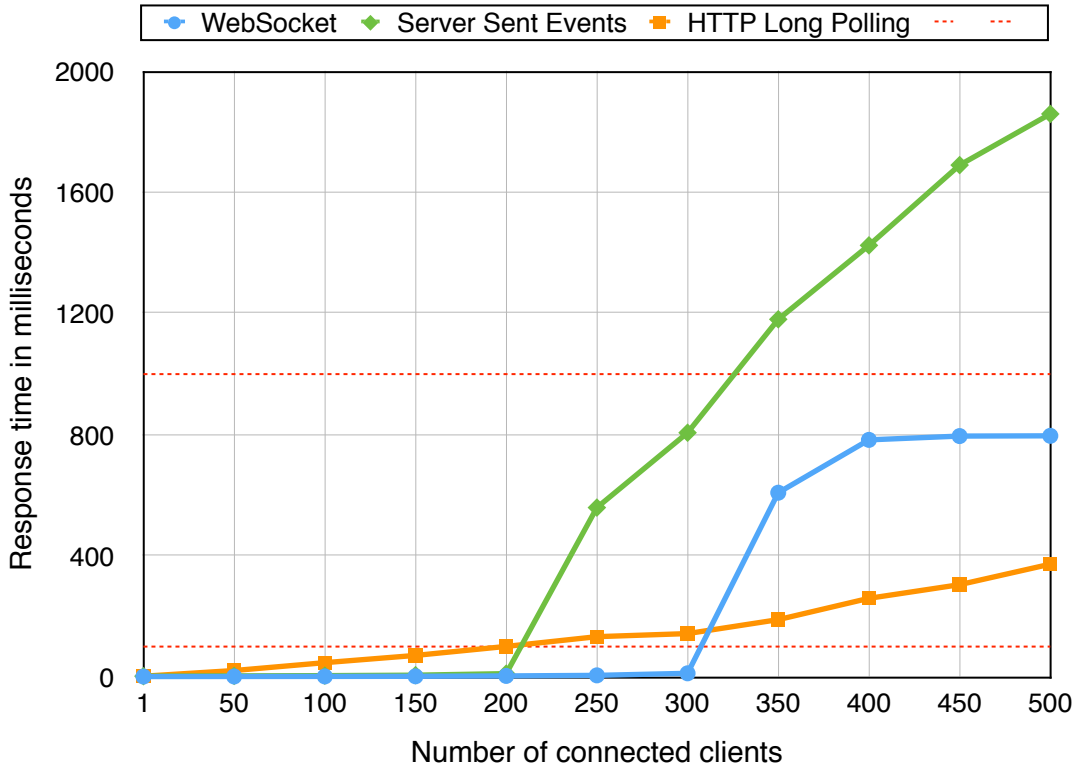

Figure 35: The response times during the first test scenario's test phase

Note: The two red dotted lines in the chart are the 1.0 and 0.1 second limits from subsection 2.9.2.

These results must be view in context with the CPU load from the previous subsection. When the CPU load is below maximum, each of the servers perform very good, clearly below the 0.1 second limit.

When the Long Polling server reaches maximum CPU load at 50 clients, we see that the response time starts to climb almost linearly with the client count.

Considering how much CPU load the HTTP Long Polling server used from the start, one could expect it to be outperformed by the other servers all the way from the start. That is also the case, but only initially. When the WebSocket and Server Sent Events servers reaches maximum CPU load at around 98% load, they really struggle to keep the response time low. With Server Sent Events it skyrockets all the way up to over 1,8 seconds, while WebSocket stays below 1 seconds at around 800 milliseconds.

Even though the sudden spikes in response time seem unnatural and anomaly like, they were encountered in all 10 test runs. Chapter 5 takes a deeper look into why this happens.

4.4 Test Phase - Scenario 2

When the second test scenario was developed, and some initial tests were run, it was clear that the response time varied a bit more than in the first test scenario. This made me calculate the the median in addition to the average response time. However, when all test results were collected, the median was not to far off from the average. Consequently I will only present the average. The response time median can be found in the Appendix.

As with the first scenario, the results here are collected from right after the chat phase is live to just before it ends.

4.4.1 CPU Load During Chat

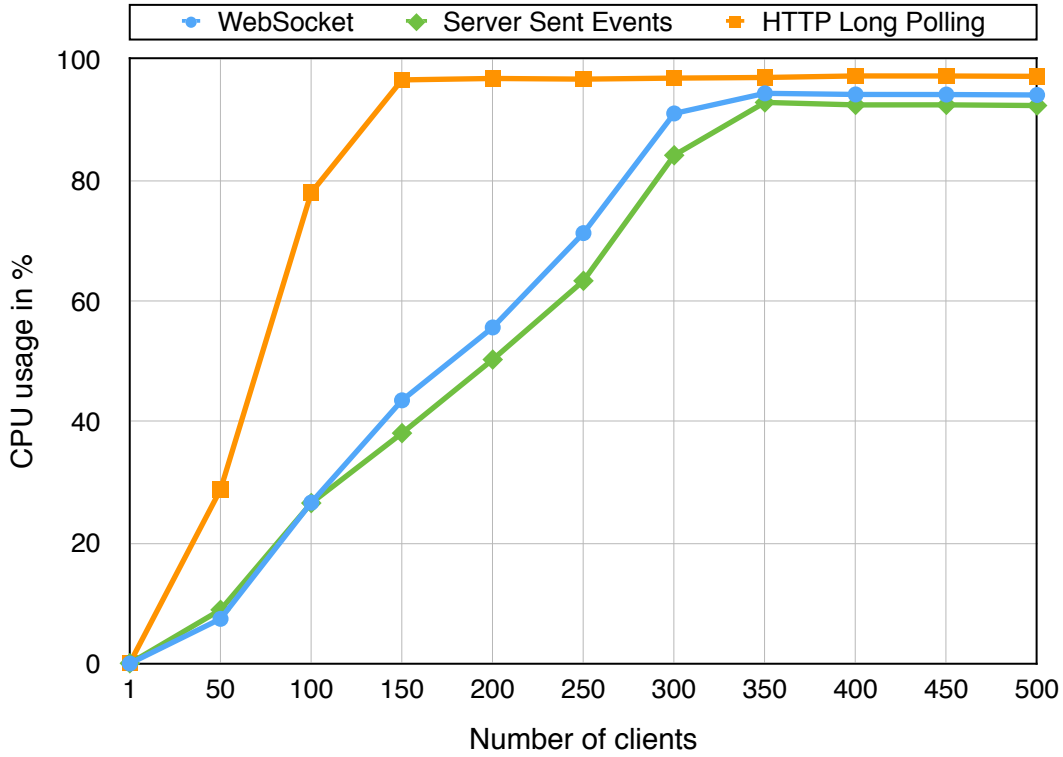

Figure 36: The CPU load during the seconds test scenario's test phase

Not too different from the first scenario, we see that the HTTP Long Polling server immediately requires a lot more CPU power than the other servers. It reaches its maximum CPU utilization with 150 clients, while the other two servers reach peak CPU usage at 350 clients.

This time around, WebSocket and Server Sent Events are very similar, with the latter being the most efficient.

4.4.2 Response Time During Chat

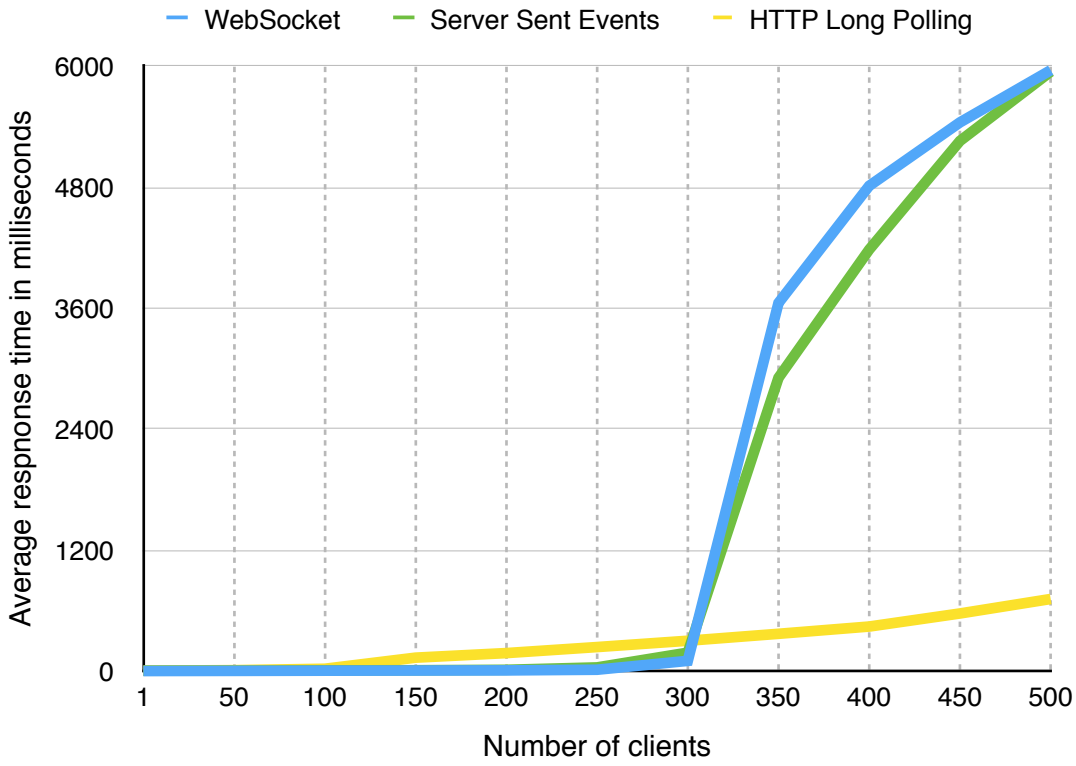

Figure 37: The response times during the second test scenario's test phase

When the Long Polling server reaches full CPU load at 150 clients we see a small jump in response time. From that point the response time increases steadily and linearly as the client count rises. With 500 polling clients, the Long Polling server responds to ping messages within 0,7 seconds.

The WebSocket and Server Sent Events servers are very similar here. They both start off with a very quick response time and it stays low as the CPU is not stressed. At 350 clients, when the CPU load reaches maximum, the response time very dramatically escalates. When there are 500 connected chat clients, the servers uses almost 6 seconds to respond.

4.5 Memory Footprint After Tests

4.5.1 Test Scenario 1

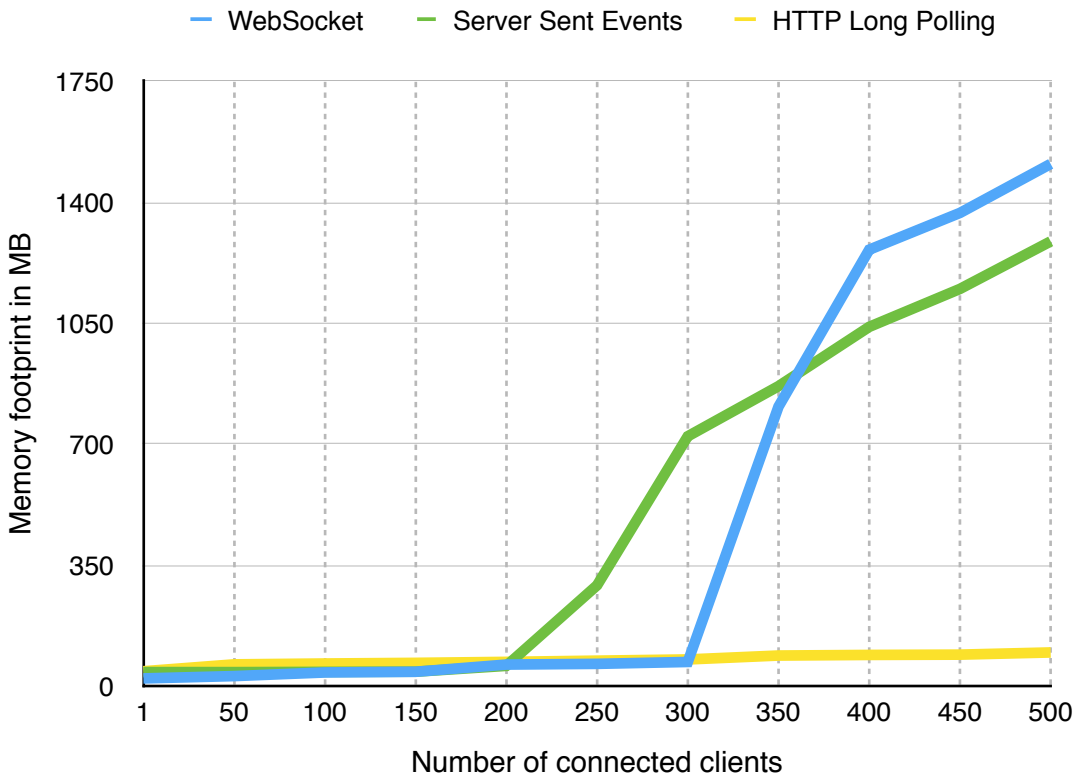

Figure 38: The memory footprint right after the first test scenario's test phase

The HTTP Long Polling server has a small and steady increase in memory consumption after the broadcast is finished. It increases linearly with the client number. That is also true for the other two servers, but only initially. Just as with the response time, the memory footprint after the tests, increases dramatically when the CPU load peaks. After the 500 client test runs, the WebSocket and Server Sent Events servers consume more than 1 GB of memory.

4.5.2 Test Scenario 2

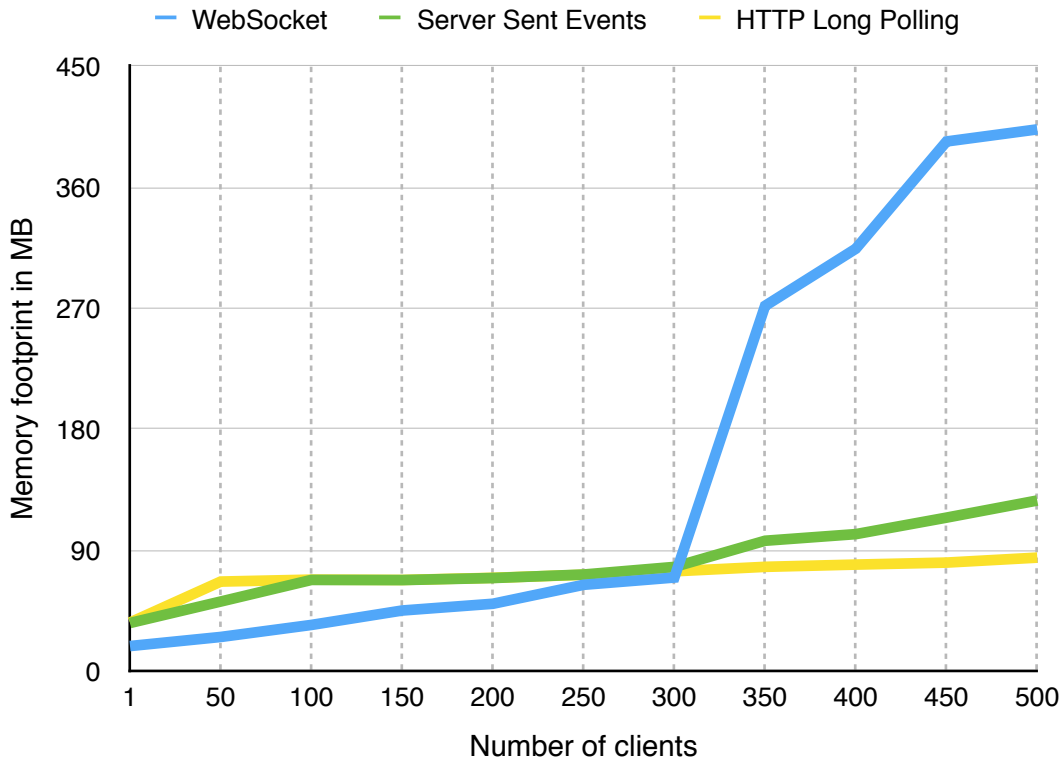

Figure 39: The memory footprint right after the second test scenario's test phase

Once again, the HTTP Long Polling server performs predictable and there is only a small an gradual increase in memory consumption. On the other hand, the Server Sent Events and WebSocket counterparts see a jump in memory footprint as they reach full CPU utilization at the 350 client mark. The jump is more significant for the WebSocket server, but definitely exists for the Server Sent Events server as well.

Chapter 5: Discussion

5.1 Introduction

The Results chapter showed that the Server Sent Events and WebSocket servers experience explosive growth in response time and memory consumption when the CPU is stressed. Because of this, I have chosen to divide the discussion into the following sections:

1. The idle client state, where the clients are connected or polling, but inactive.
2. The load testing part, where the server is not under maximum stress levels.
3. The stress testing part, where the server is under great levels of stress with maximum CPU load.

The most interesting results come from the stress testing, where we see some anomalies with regards to excessive memory usage and sudden spikes in response times. These unexpected results needs explanation, and I will try to clarify them with possible solutions.

Lastly, to see how the three different transports compare in terms of performance, I will look more into how usable they are, from a programmers perspective.

Before analyzing the results, Jakob Nielsen's 3 Important Limits.

5.2 Response Times: The 3 Important Limits

This heading is the name title of an article[26] 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."

The 10 second limit can be discarded, as none of my test runs exceeded it. The remaining two limits form a great base for what is acceptable results in a real-time setting. Of course it varies from application to application which one of these limits is kept. For a chat application it is probably not too important that a message arrives within 0.1 seconds, but it would be nice not having to wait 10 seconds. 1.0 seconds for a chat application seems reasonable. There are other types of applications where the 0.1 second limit seems a better fit. Let us say you control a remote camera or even a remote controlled vehicle from a website. When controlling that vehicle, 1.0 second delay after hitting controls would seem like an eternity.

5.2 Idle Clients

5.2.1 CPU Load

When looking at the results found in subsection 4.2.1, all three transports perform very good, keeping the CPU usage low, even as the client count increases to 500. Even though the three different servers all perform nicely, the WebSocket server uses less than half the CPU cycles the other two servers does. Never once does it exceed 1% of CPU use, while the Long Polling and Server Sent Events servers always lies between 2 and 3 percent. The fact that the HTTP Long Polling and Server Sent Events servers perform very similar here can be explained by the fact that they both run on top of the same Express[39] server.

These are certainly good results, but in fact they are totally expected. Even though 500 standalone clients was the maximum my client computer could handle, 500 connections is not a lot for a server to handle.

5.2.2 Memory Footprint

Here we look at the results found in subsection 4.2.2.

The memory footprint of the three servers start out differently. The WebSocket server starts with a memory footprint of 17MB for 1 user, while the Long Polling and Server Sent Events servers both start out at 28MB. Once again, the similarities between the two latter mentioned servers can be explained by the case that they both use the same web application framework, Express. The WebSocket server on the other hand uses the very bare bones WebSocket library ws[36] and it clearly has a very small memory footprint.

As the number of connected or polling clients increases, the WebSocket server's memory footprint increases faster than the other two servers and around 400 clients it catches up to the other two servers. This is expected as WebSocket is a stateful protocol, requiring a larger memory footprint for each connection. While, this is also to some degree true for Server Sent Events, the "connection" is more lightweight. The penalty in memory consumption by using WebSocket is visible, but not too expensive. Overall, these results are expected.

5.2.3 Response Time

The discussion here is aimed towards the results presented in subsection 4.2.3.

The response times for all three servers is really good here and always way below the 0.1 second limit from Nielsen. Once again we see the HTTP Long Polling and Server Sent Events versions are very close to each other while the WebSocket version outperform them both. The WebSocket server always respond within 1.3 to 1.5 milliseconds, while the other two generally uses 3 milliseconds. The similarities between the Long Polling and Server Sent Events servers can once again be explained by the common Express server, and also by the fact that these two tests use the same HTTP based ping client. In contrast, the WebSocket server uses a different standalone WebSocket based ping client. Once again, we see the WebSocket server come out on top.

5.3 Load Testing

The discussion in this section is aimed towards the *load testing part* of the test. That means the part of the results where the CPU utilization is below maximum. Because some of the servers started to behave unexpected during stressing load levels, the separation between load and stress tests have been created. The result analysis from the stress test is found in section 5.4.

5.3.1 Test scenario 1

In this subsection I look into the response times for the first test scenario during its broadcast phase. Figure 34 shows the same results found in subsection 4.3.2, but this time with the 1.0 and 0.1 second limits by Nielsen.

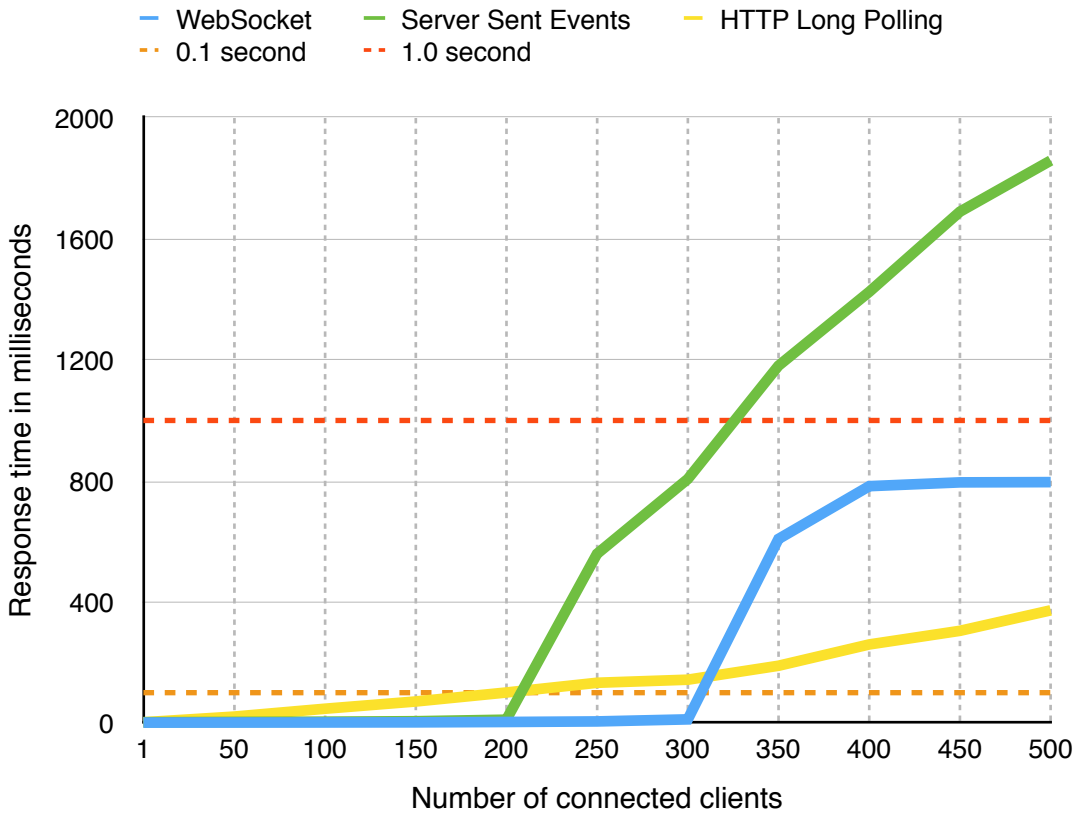

Figure 40: The response times during the first test scenario's test phase

Immediately, it looks like Long Polling severely outperforms the WebSocket and Sever Sent Events versions. However, since this section only looks into the load test part of the results, we must ignore the part where the server is stressed.

Figure 35 shows the same results, this time zoomed in to show a maximum of 110 milliseconds on the Y axis and 200 clients at the X axis. With 250 clients, the Server Sent Events server met its break point - and the load test turned to a stress test. One could argue that the HTTP Long Polling server is always being stressed as the CPU utilization reaches 98% with just 50 clients, but still the increase in response time grows gradually and not sudden as with Server Sent Events and WebSocket.

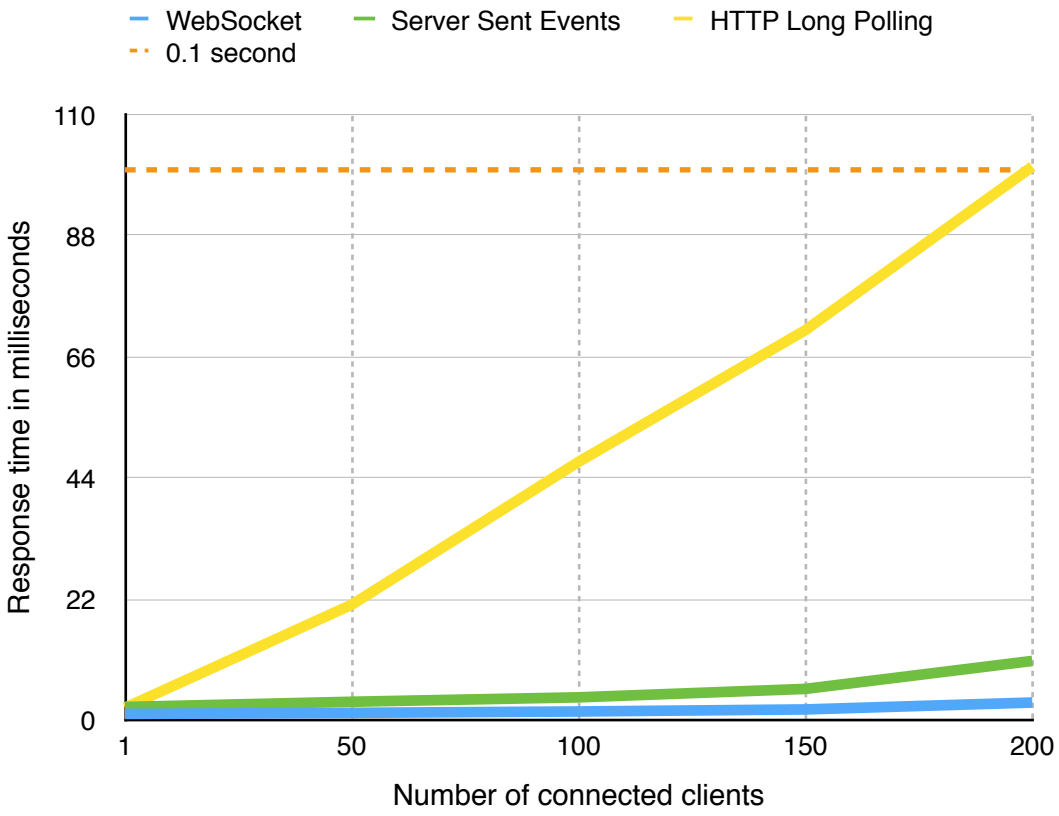

Figure 41: The response times during the first test scenario's test phase

Here, a totally different picture emerges. Here both WebSocket and Server Sent Events totally dominates the Long Polling server in terms of response time, and you can see them staying almost flat and always way below the 0.1 limit from Nielsen. The HTTP Long Polling server has an gradual and almost linear growth in response time.

These results are expected. WebSocket should outperform both the other two servers and, while being stressed, the Long Polling version sees an almost linear growth as the client count increases. The fact that the Long Polling version reaches stressing levels way before the other two servers is also expected. As discussed in the background chapter, HTTP is not designed for this type of real-time behavior, requiring a request for each server update.

From Kristian Johannessen's thesis Server Sent Events was expected to perform quite comparable[2] to WebSocket, and that is the case with these results. Being based on HTTP and not being a new standalone protocol, Server Sent Events performs very well.

5.3.2 Test scenario 2

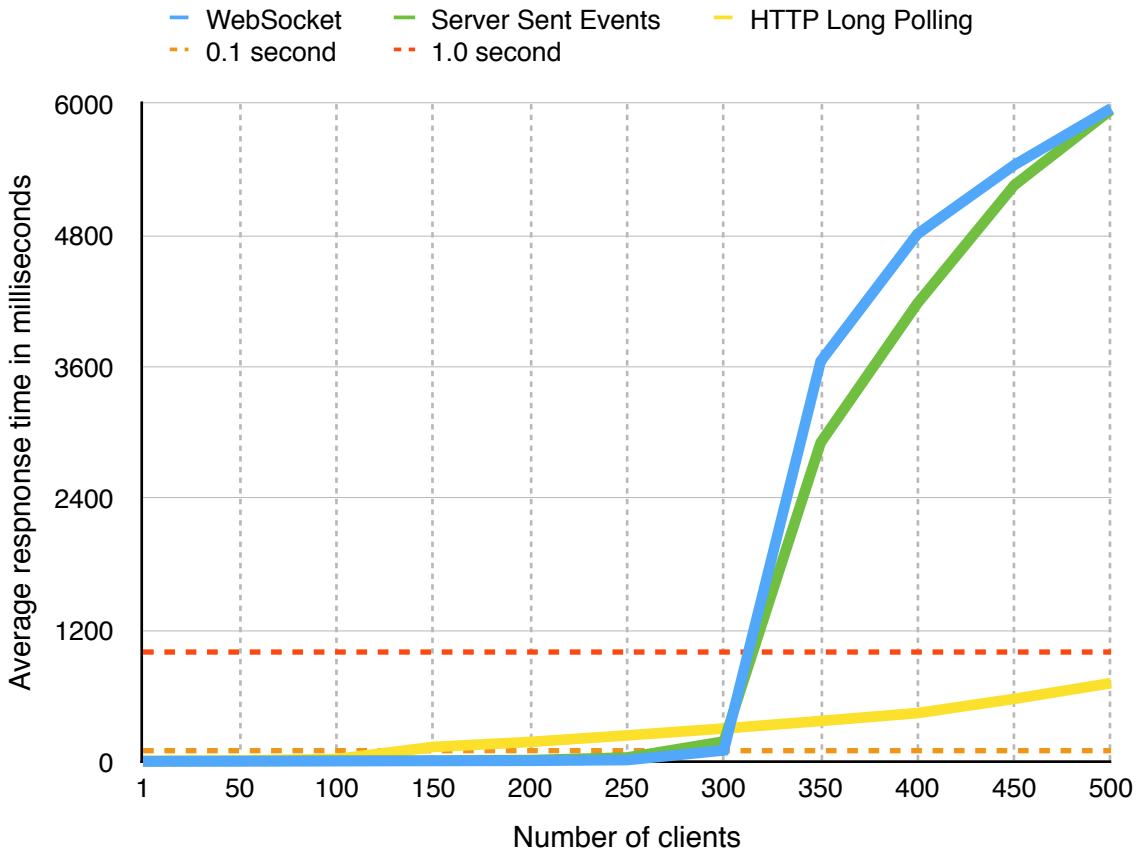

Figure 42: The response times during the second test scenario's test phase

The graph in figure 36 is the exact same graph that was presented in subsection 4.4.2, but this time with lines representing the 0.1 and 1.0 second limits by Jakob Nielsen. Straight away, it once again looks like if Long Polling devours the other two servers in terms of response time. That is true as well, and even more so this time than in the first scenario. However, as this section looks into the load part, not the stress part, we must cut out the part where CPU utilization is at a maximum.

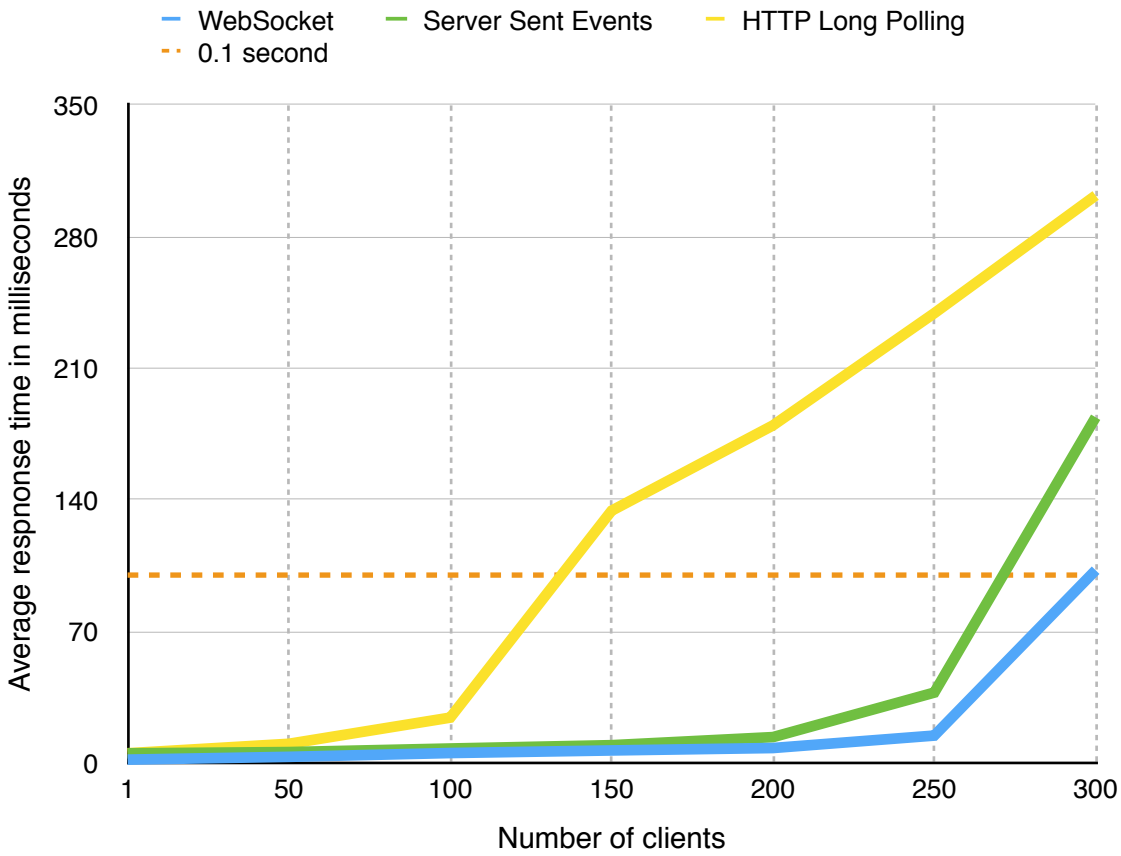

Figure 43: The response times during the second test scenario's test phase

Both the WebSocket and the Server Sent Events servers reaches their maximum CPU utilization at the 350 connected client mark, so I have decided to cut out the part with 350 and more clients. Once again, it must be noted that the HTTP Long Polling server reach its maximum CPU load ahead of the other two servers, at 150 clients. Even though it does, it doesn't seem to have the same explosive growth in response time as the other two servers. As with the first test scenario, this one shows how much better the Server Sent Events and the WebSocket servers perform compared to the HTTP Long Polling counterpart, when the load is less than full. The two best performers are almost identical up until 200-250 clients, where we see them diverge a bit. At this point it is becoming apparent that they are affected by the high levels of load and the response time begins to increase rather fast.

The fact that HTTP has no built in mechanism for real-time behavior and the increase in architectural complexity for the Long Polling server (see subsection 3.7.4), made me expect it to be outperformed. What I didn't expect to see though, was how well the Server Sent Events server performed. I expected WebSocket to be the clear winner here, as the protocol works bidirectionally, but even with the Server Sent Events server requiring a separate HTTP route for the incoming messages, it performed very well.

5.3.3 Load Test Summary

- The HTTP Long Polling server reaches maximum CPU load way before the other two servers and the response time grows linearly as the client count increases.

- When the Server Sent Events and WebSocket server reaches full CPU utilization, their response times goes through the roof. This is seen as an anomaly, and will be discussed in the two following sections.
- As long as the load levels are below maximum, all three servers perform quite well, always staying below the 0.1 second Nielsen limit.
- As expected Server Sent Events performed well in the first scenario, as the messages are server-to-client only.
- Unexpectedly, Server Sent Events performed very good in the second scenario, even though it required a second HTTP route for incoming chat messages.
- WebSocket is the real winner here, having the lowest response times throughout the whole load tests. This was expected.

5.4 Stress Testing

5.4.1 Test Scenario 1

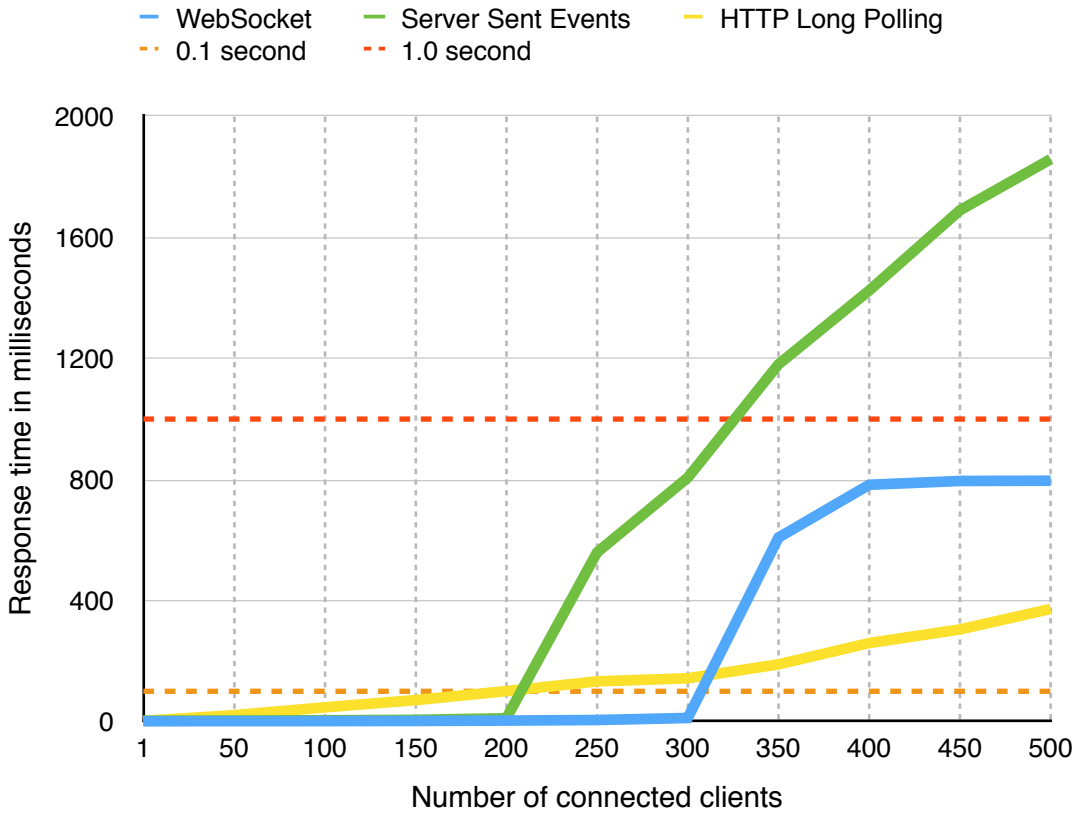

Figure 44: Response times during the broadcast phase in test scenario 1

The Long Polling server was expected to reach max CPU utilization before the other two servers because of its architectural complexity. That expectation was correct, with the HTTP Long Polling server reaches its CPU utilization peak at just 50 clients, while the Server Sent Event and WebSocket servers reaches CPU peak at 250 and 350 respectively. It was also expected that WebSocket would perform better than Server Sent Events overall, as the protocol was built to be efficient. But the way the different servers performed after reaching CPU peak, was totally unexpected.

The HTTP Long Polling server has a gradual and steady increase in response time as the client count increases. It performs actually really well and stays below the 1.0 second Nielsen limit at all times. It does however breach the 0.1 second limit at the 200 client mark.

The Server Sent Events server performs, as previously stated, great while the load is moderate, but when CPU peak is reached, the picture changes quickly. The response time explodes to the roof, and at the 350 client mark, breaks the 1.0 second limit. With 500 connected clients, the Server Sent Events server use a whopping 1,8 seconds to answer the ping client.

Similar to the Server Sent Events server, the WebSocket counterpart also experience an explosive growth in response time when the CPU is brought to stress levels. Interestingly the response time seem to stabilize just under 800 milliseconds ensuring it stays below the 1.0 second limit throughout the whole test scenario.

The sudden spikes in response time found with the Server Sent Events and WebSocket servers were not predicted. A more gradual and almost linear growth in response time, like with the Long Polling server, was expected. Plausible explanations for these anomalies will be discussed in section 5.5.

5.4.2 Test Scenario 2

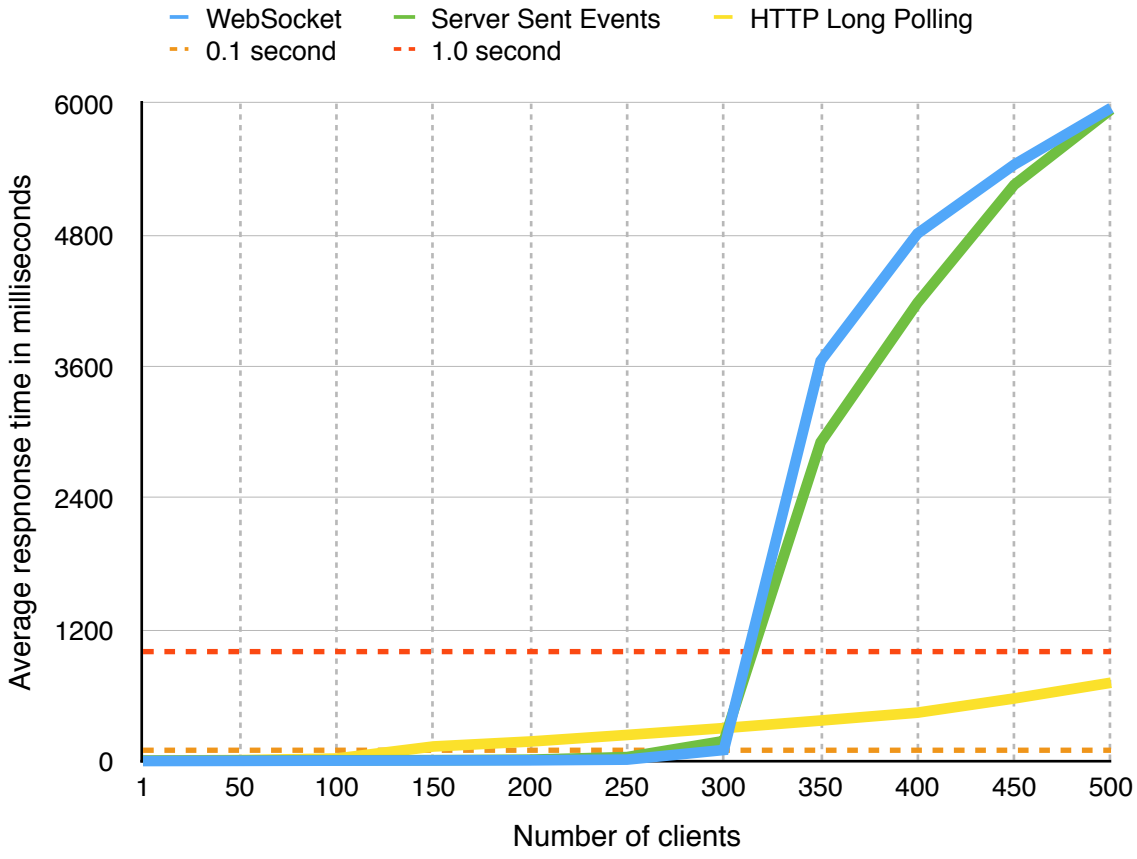

Figure 45: Response times during the chat phase in test scenario 2

It was expected that the HTTP Long Polling server would reach maximum CPU utilization before the other two servers. This proved to be an accurate expectation, as it reached full CPU use with 150 clients, while the Server Sent Events and WebSocket servers managed to reach 350 clients before seeing the same CPU load levels.

Just as with the first test scenario, it was assumed that the increase in response time when stressing the server, would be linear. Once again it proved to be right for the HTTP Long Polling server, while the other two servers, saw their response time increase dramatically when stressed and skyrocket way above the HTTP Long Polling equivalent. In this scenario it is only the Long Polling server we see stay below the 1.0 second Nielsen limit. The other two servers reaches response times of nearly six times that limit.

Once more, it must be stated that these results are very interesting, and maybe more surprising than the results for the first scenario. The chat application was a perfect fit for WebSocket with native bidirectional messaging support. The Server Sent Events counterpart was architectural more complex and the Long Polling version even more so. The increase in architectural complexity does not seem to handicap the servers, as the worst performer here is the WebSocket one.

5.4.3 Stress Test Summary

- Once the Long Polling servers are stressed, the response times behaves expectedly and increases linearly with the client count. In both scenarios, they breach the 0.1 second Nielsen limit while stressed, but always stay below the 1.0 second limit.
- Both the WebSocket and Server Sent Events servers experience an unexpected explosive growth in response times when the CPU is stressed. A linear Long Polling like growth was expected as the client count increases. The sudden and dramatic increase must be considered an *anomaly* and can be an issue with the software platform. Possible explanations will be presented in the following section.
- The anomalies cannot be ignored, pointing to a clear win for HTTP Long Polling.

5.5 Memory and Response Time Anomalies - Possible Issues With the Software Platform

I did not focus much on memory when implementing the test scenarios, but decided to record memory consumption before and after the test, to see if there were any unexpected results - anomalies. 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 the 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.

Following in figure 40 and 41, 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.

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.

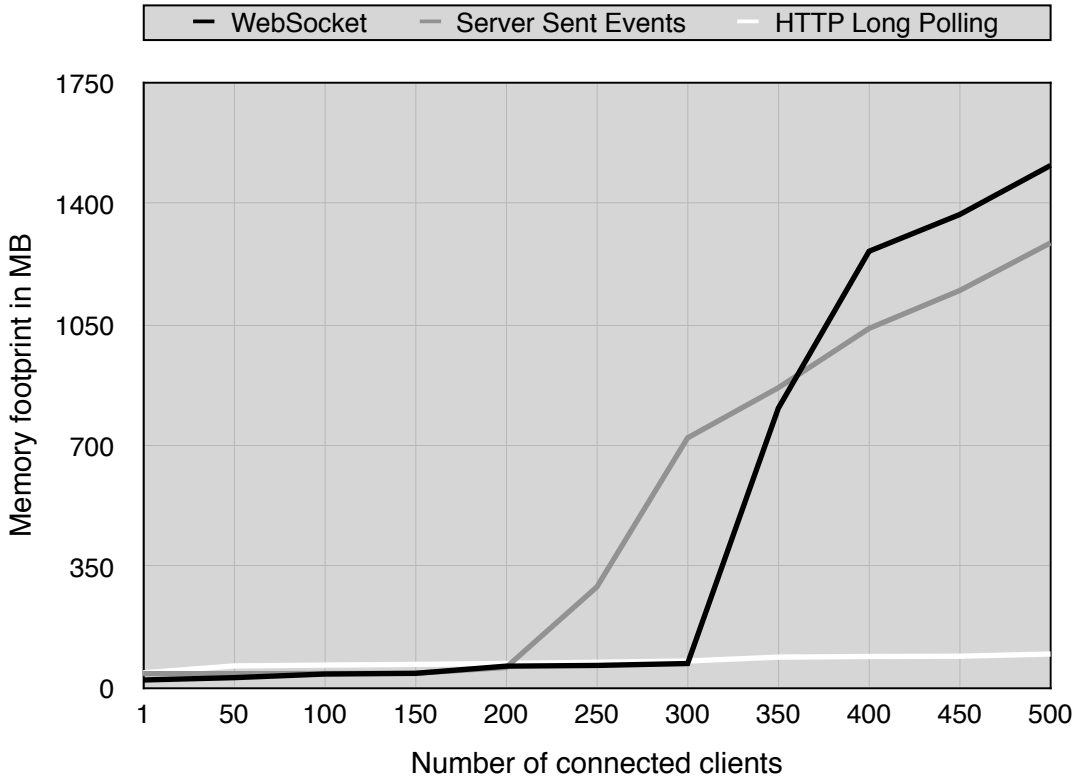

Figure 46: Memory footprint right after the broadcast phase in test scenario 1

Figure 47: Memory footprint right after the chat phase in test scenario 2

The dramatic increases in memory footprint we see here are comparable to the sudden escalation we see with the response times and both happen when the CPU is stressed very hard. 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. Consequently the possibility of a bug in Node.js or one of the libraries in use became a reality.

For the rest of this section, I will list and discuss possible explanations to why these unanticipated spikes occur.

5.5.1 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 explain the high response times and large memory consumption during high load, at least for the WebSocket version.

5.5.2 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). Why then would the increase in response time and memory footprint only happen with the Server Sent Events version?

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 the same updated V8 engine. Maybe this new engine running in io.js or Node.js 0.12 fixes the test anomalies.

5.5.3 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.

5.5.4 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.

5.6 Implementation

Up until this point, all discussion has been related to the test results. How different technologies compare in performance is of course very important, 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

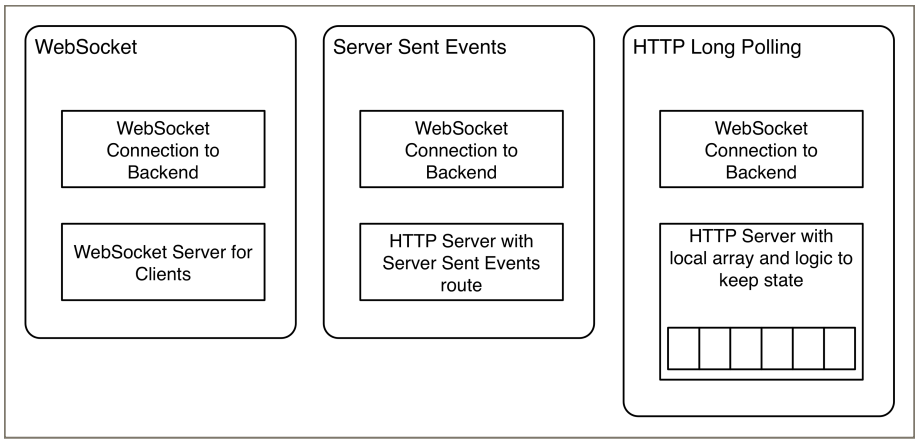

Figure 48: The three different servers in the first test scenario

Figure 42 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 WebSocket 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 were 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. As seen in figure 6 in subsection 2.4.3, the Long Polling server must locally store each broadcast message to ensure that all clients receive them. This means a quite substantial increase 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

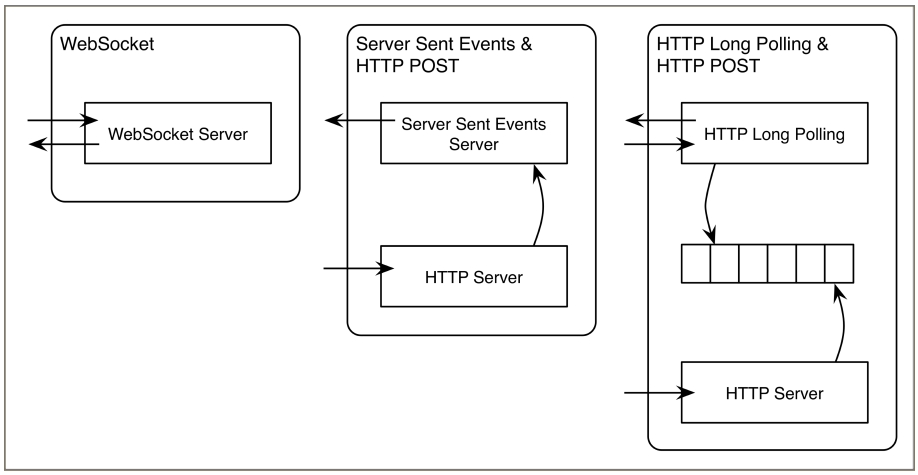

Figure 49: The three difference servers in the second test scenario including arrows for incoming and outgoing messages.

Looking at figure 43, 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 get them all. Conceptually more complex and more difficult to develop.

5.5.3 Summary

- If there only is a need for outgoing server messages, Server Sent Events is just as simple to use as WebSocket.
- If the is a need for both outgoing and incoming messages, WebSocket is clearly the easiest pick, as the protocol is full-duplex by nature. However Server Sent Events plus an additional HTTP route for incoming messages is also easy to grasp.
- Using only HTTP is conceptually more complex and requires some work around to make up for the fact that the protocol is stateless.

5.7 Thesis Conclusion

Her svarer jeg direkte på problemstillingene. 1 side er nok.

5.8 Further Work

HTTP 2.0
WebRTC
Samme oppgave på flere software platformer

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. Berners-Lee, T. The HTTP Protocol As Implemented In W3. 1991; Available from: http://www.w3.org/Protocols/HTTP/AsImplemented.html.

7. Ltd., T. HTTP Scoop. 2014; Available from: http://www.tuffcode.com.

8. Garrett, J.J., Ajax: A New Approach to Web Applications. 2005.

9. W3C. XMLHttpRequest Level 1 - W3C Working Draft. 2014; Available from: http://www.w3.org/TR/XMLHttpRequest/.

10. T. Bray, E., The JavaScript Object Notation (JSON) Data Interchange Format.

11. 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/.

12. HTML5. 2014; Available from: http://en.wikipedia.org/wiki/HTML5.

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-eventsource-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. Google. V8 JavaScript Engine. Available from: https://code.google.com/p/v8/.

24. Dahl, R., JSConf.eu: Node.js. 2009.

25. J. D. Meier, C.F., Prashant Bansode, Scott Barber, Dennis Rea, Performance Testing Guidance for Web Applications. 2007.

26. Nielsen, J., Response Times: The 3 Important Limits. 1993.

27. PayPal. Node.js at PayPal. 2013 13.04.15]; Available from: https://www.paypal-engineering.com/2013/11/22/node-js-at-paypal/.

28. Sharp, R. nodemon. 2015 13.04.15]; Available from: http://nodemon.io.

29. GitHub. Most starred repositories. 2015 13.04.15]; Available from: https://github.com/search?utf8=✓&q=+stars%3A%3E1000.

30. npm. Node Package Manager. 2015; Available from: https://www.npmjs.com.

31. Apple. JavaScript for Automation Release Notes. 2014 13.04.15]; Available from: https://developer.apple.com/library/mac/releasenotes/InterapplicationCommunication/RN-JavaScriptForAutomation/.

32. Crockford, D., JavaScript: The Good Parts. 2008: O'Reilly.

33. Joyent. Readline Node.js. 2015 13.04.15]; Available from: https://nodejs.org/api/readline.html.

34. Texin, T. Unicode Supplementary Characters Test Data. 2010 [cited 2015 12th january 2015]; Available from: http://www.i18nguy.com/unicode/supplementary-test.html.

35. Qveflander, N., Pushing real time data using HTML5 Web Sockets. 2010.

36. Stangvik, E.O. ws: a node.js websocket implementation. 2014 23.03.15].

37. Stangvik, E.O. websocket client benchmark. 2015; Available from: http://einaros.github.io/ws/benchmarks.html.

38. Hellesøy, A. EventSource-node. 2015 13.04.15]; Available from: https://github.com/aslakhellesoy/eventsource-node.

39. StrongLoop. Express - Node.js web application framework. 2015 [cited 23.03.15]; Available from: http://expressjs.com.

40. Request - Simplified HTTP client. 2015 [cited 13.04.15]; Available from: https://github.com/request/request.

41. Modulus. process-monitor. 2015 [cited 13.04.15]; Available from: https://github.com/onmodulus/process-monitor.

42. Codenomicon. The Heartbleed Bug. 2014 [cited 22.03.15]; Available from: http://heartbleed.com.

43. Google. Chrome V8 Design Elements. 2012 [cited 15.04.15]; Available from: https://developers.google.com/v8/design.

44. StrongLoop. Node.js Performance Tip of the Week: Event Loop Monitoring. 2014 [cited 15.04.15]; Available from: https://strongloop.com/strongblog/node-js-performance-event-loop-monitoring/.

45. Kazemier, A. Changes in ws 0.5. 2014 [cited 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 [cited 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

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

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

Test Results

Idle CPU Load

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**

Tekst

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**

Tekst

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**

Tekst

Idle Memory Footprint

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**

Tekst

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**

Tekst

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**

Tekst

Idle Response Time

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**

Tekst

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**

Tekst

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**

Tekst