WebSocket Scalability

Comparing scalability of real-time mechanisms

Abstract

Text

Contents

Abstract 2

Contents 3

List of figures 5

List of examples 6

Preface 7

Acknowledgements 8

Introduction 9

Background 12

Introduction to the Web 13

HTTP 14

Ajax, HTML5 and Web Apps 16

Real Time HTTP 18

WebSocket 21

Server Sent Events 26

Node.js - JavaScript Everywhere 28

Performance testing 30

Project Part 1 31

Introduction 32

Methodology 33

Results 42

Project part 2 54

Methodology 55

Test environment 56

Results 57

Analysis 58

Analysis heading 1 59

Conclusion 60

Further work 61

Appendix 62

Bibliography 63

List of figures

Text

List of examples

Text

Preface

Text

Acknowledgements

Text

Introduction

Real-time and the Web

To understand what is meant by the term real-time in this thesis, consider the following example. We have 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. This immediate handling of new data on the server, as well as a bidirectional pipe that makes it possible, is what real-time means in this thesis. In fact, this example is very similar to one of the methods used in the project part later.

For many types of applications having this sort of real-time behavior is crucial. For systems where uptime is critical, health related systems for example, real-time error message delivery can be a matter of life and death.

As will be emphasized in the background part of this thesis, the web is inherently not designed for real-time, as HTTP, the protocol powering the web, is unidirectional and request-response oriented. However, with HTML5, the W3C finally brings native ways to build real-time apps for the web. WebSockets and Server Sent Events are the technologies that will be explored in this thesis.

Problem statements

Does WebSocket perform and scale better than traditional HTTP methods for real-time server-push?

When we want to achieve real-time behavior in our web app, how well does WebSockets really scale? Is there a significant difference to HTTP Long Polling and Server Sent Events? Metrics that come into play here; server CPU load, memory footprint and network usage.

Under what load levels does WebSocket break? Is this level higher than for HTTP Long Polling or Server Sent Events?

How far can we push WebSockets before we see a significant drop in response time from the server. Then compare this to the break point for HTTP Long Polling and Server Sent Events. Is there a significant difference? Most interesting metric here is the number of messages the server can handle per second.

Does client-to-server WebSocket messages perform and scale better than HTTP POST requests?

One of the powers of WebSockets is the ability to push messages both from the server to client and client to server. Although HTTP POST messages are similar to WebSocket client to server messages, it is interesting to find out how much better, if better at all, WebSockets scale compare to HTTP with client-to-server messages. Metrics that come into play here are; server CPU load and memory footprint, network usage and most important - response time.

Stress test client-to-server messages to find out the load levels when WebSocket break.

Under what levels of load does WebSocket and HTTP POST messages start to show large deviation in response time? Does WebSocket provide better scaling capabilities compare to HTTP? Interesting metric here is number of messages per second the server can handle.

Compare the most widely used web browsers and see how they handle huge amounts of data sent from the server using HTTP and WebSockets.

Load test the newest versions of the most common browsers. Web browsers to examine: Mozilla Firefox, Google Chrome, Apple Safari, (Microsoft Internet Explorer). Metrics: CPU load and memory footprint.

Related Work

Not so much. WebSockets are quite new.
Kristian
Joevik

Kristian Johannesen

Terminology

Transports	Protocol on the transport layer. In this thesis it's used in context with either WebSocket, Server Sent Events or HTTP Long Polling

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

Where I've put the source code.

Outline

Text

Background

Introduction to the Web

Tim Berners-Lee understood the potential of networked computers when he invented the World Wide Web in 1991, but he cannot have had any idea of the impact his invention would have on the world. Today, at the age of only 23 years, the web is hard imagining living without. It's remarkable how quickly the technology has merged itself into the society, even for non-tech-savvy people. We do everything from reading news and paying bills to social interaction and playing games on the web. Even though the web is just a subset of the internet, many people today don't know the distinction.

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 an 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's 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 will be revealed in this thesis, it is hard and suboptimal to develop real-time apps using HTTP.

As mentioned earlier, 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 set out to solve most of HTTP's limitations compared to TCP.

Following in the background part of this thesis, is a presentation of HTTP, Server Sent Events and WebSocket. This will form a good foundation for understanding what comes later in the thesis.

HTTP

HTTP is an application level network protocol used to deliver data from a server to a client. The original definition of HTTP was written in 1991 by Tim Berners-Lee, shortly after he invented the Web at the physics institute in CERN. The web was intended as a platform in which researchers at CERN could easily share and access each others documents. HTTP was originally designed to be a very simple protocol with just one purpose, to fetch (or GET) documents. The basic principle behind HTTP is therefore just a single request sent from the client to the server and the single requested document sent back as a response. There wasn't any need for storing information about the clients on the server side, so the protocol was designed as a stateless protocol. Every connection is treated the same way.

The whole request is sent as plain ASCII text and consists of a GET followed by the document's address. The server parses the GET request and sends the the requested document back as a response. The server terminates the underlying TCP connection when the document is sent.

```
GET /index.html HTTP/1.0
Host: www.uio.no
<blank line>
```

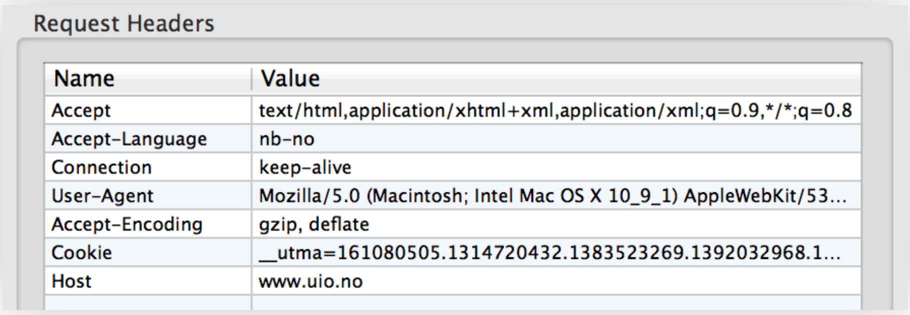
Figure X: A GET request to www.uio.no. The blank line indicates end of message.

The HTTP description above is a brief run through of Tim Berners-Lee own words about version 0.9 of the protocol, from 1991[1].

The web's potential, also outside of the research world, was clearly huge and on the 30th of April 1993, it was decided that the web was to be «freely usable by anyone, with no fees being payable to CERN»[2]. A year later, in October 1994, The W3C (World Wide Web Consortium) was formed as the main standards body for the Web and to this day Sir Tim Berners-Lee is still the director. In 1996 the W3C introduced the finalized version of the protocol, HTTP/1.0. Version 1.0 added two methods to the already existing GET method - POST and HEAD. POST is used in conjunction with web forms and used when a user wants to submit data to the server, and HEAD is used as a GET where you don't want the actual response data, but only the response *header fields*.

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 X 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's useful for the server to know what kind of language the client wants the requested document in.

The image below shows the entire GET request my web browser initiated when going to www.uio.no, with the browser adding several header fields:

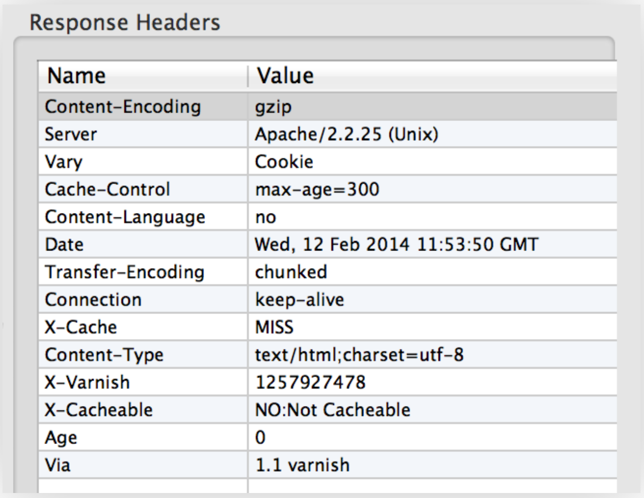

Request Headers

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

Figure X: HTTP GET request to www.uio.no. Captured using HTTPScoop[3].

As you can see above, there's 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 on, while the Accept header tells the server what files the client web browser can read. HTTP Headers are an important part of the protocol and adds a marginal degree of state.

The server responds to the request with a HTTP response. The response embody header fields followed by the HTML code. 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. Most files on the website change only occasionally, so chances are that your browser has cached most of a site's content, so that the next time you visit, the number of necessary GET requests are way lower. Below is an example of a server response with its headers:


Response Headers

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

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

Considering that http://www.uio.no is a typical web site, it doesn't require much insight to realize that closing the underlying TCP connection after each server response is very inefficient. When HTTP/1.1 arrived in 1997 this issue was resolved by allowing the client to tell the web server to keep the lower level TCP connection open. This is done by providing the «Connection: keep-alive» header as seen in Figure X. Many years on, HTTP/1.1 is still the current version.

Ajax, HTML5 and Web Apps

As mentioned, HTTP was designed to serve static hyperlinked documents. Today however, you only sporadically visit a site that is static and pure HTML. Almost every site you visit are highly interactive and complex creations 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 of 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. Today they are lightning fast.

But, how could developers build these Web Apps with the limitations the web forced upon them? Central to this development was Ajax, a technique for fetching data in the background using JavaScript and HTML5.

HTML5

HTML5 is the fifth revision of the HTML markup language and the first major update since HTML4 was standardized in 1997[4]. Even though HTML5 adoption stated many years ago, the W3C recommendation was just recently finalized[5]. HTML5 is, despite its name, much more than just an updated HTML version. It's a collection of many technologies that's 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.

The new features include:
- Several new HTML markup tags, including audio, video, canvas and svg tags for native, audio, video, 2D graphic and vector graphic support respectively.
- Version 3 of the CSS styling language with support for animations, 2D and 3D transitions, media queries to support multiple screen sizes and more.
- New JavaScript APIs that include support for Geolocation, Camera, Microphone, Offline apps, Web Storage, Server Sent Events and WebSockets.

Prior to HTML5, accessing a device's hardware, like camera, microphone or GPS was not possible without proprietary web plugins, like Adobe Flash. With HTML5 it's possible to build applications that resemble their platform native counterpart, right in the browser. With tools like PhoneGap[6] and Cordova[7], it's possible to take it one step further and package web apps to run along side native applications on the targeted device. As touched upon earlier, a great deal of mobile apps today are really web apps running in a WebView (an embedded web browser window) packaged by Cordova or PhoneGap. With HTML5 the term *web apps* truly comes to life.

Ajax

Ajax as a term was introduced by Jesse James Garret in 2004, when he wrote "Ajax: A New Approach to Web Applications"[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, all using existing HTTP methods such as GET and POST. Previously, a web browser typically requested a whole 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 center of web apps, and it brought interactivity to an otherwise static web. Even though Ajax gave new possibilities, there was no going around that Ajax didn’t solve HTTP’s problems, only improved an otherwise unsatisfactory situation.

Real Time HTTP

For many applications pushing data between server and client is essential. Let's 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 are 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, right? Following are a set of techniques, that accomplishes server push using HTTP.

HTTP Polling

The first solution that comes to mind is having timer running in the client side JavaScript, that periodically 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 not a great thing, as it congests the network with unnecessary messages.

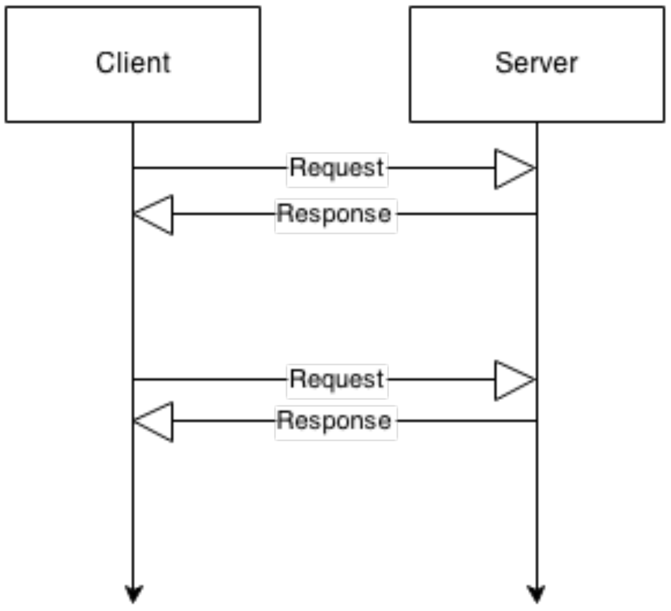


Figure X: HTTP Polling example

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 only using existing HTTP technologies. The two most used once are Long Polling and 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[11]. Long Polling gives the user the impression of having data pushed from the server, even though it in theory is not.

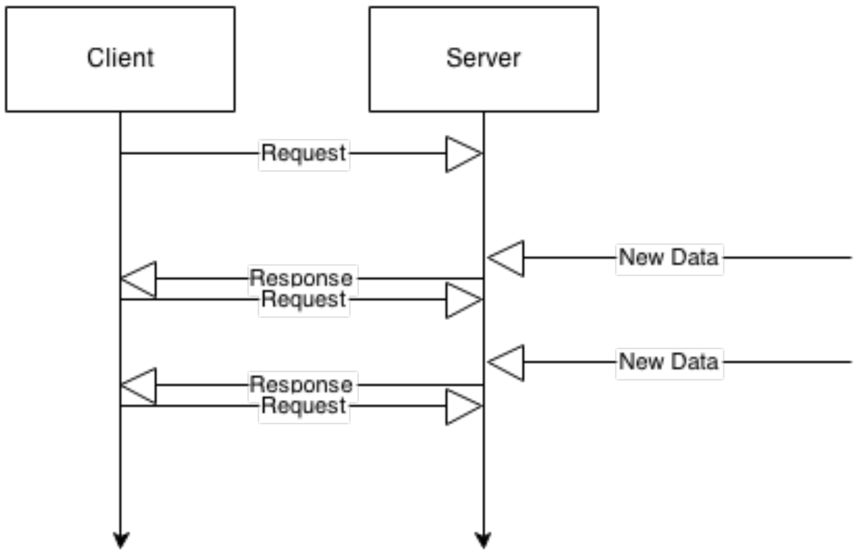


Figure X: HTTP Long Polling Example

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.

Why Comet and HTTP is Unsatisfactory for Networking

So, 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 script tags 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.

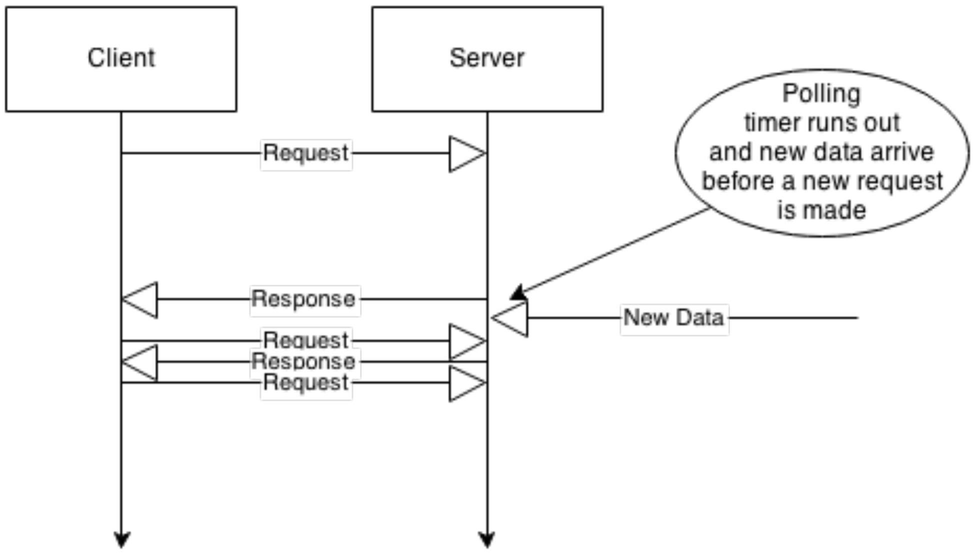

Client
Server
Polling timer runs out and new data arrive before a new request is made
Request
Response
New Data
Request
Response
Request

Figure X: 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 earlier GET example to www.uio.no, 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*. For many real time applications where you want to send small messages, maybe just a couple of bytes, this wast amount of unnecessary header data is repeated for each packet and could cram the network.

Lastly, let's not forget that developers of native applications have had powerful full duplex TCP sockets forever. This is a feature web developers just doesn't have with HTTP.

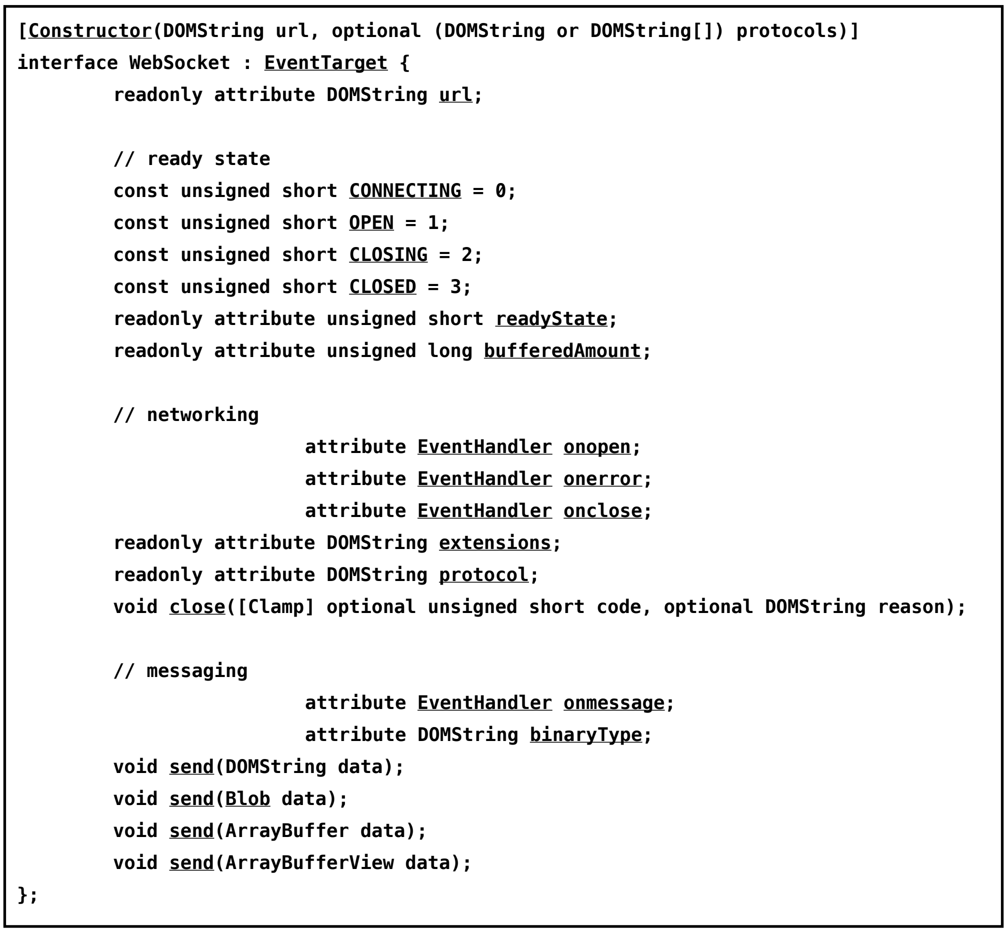
Even though Long Polling and HTTP Streaming accomplishes push behavior, there are, as you have seen, several downsides to using the two techniques. Compared to the lower level more powerful TCP sockets, building real time networked applications for the web introduces many obstacles. Many types of real time applications that would benefit hugely from having a fast lower level TCP like protocol.

WebSocket

WebSocket is meant to be the TCP like protocol the web clearly needed to evolve as a rich application platform. It promise to be all about performance, simplicity, standards and HTML5[12] 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[13]) and the API.

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 X: The WebSocket API.

States

In contrast to HTTP, WebSockets are not stateless and the API lets us access a WebSocket's state with the attribute readyState. A socket can here have 4 different states during it's lifetime: *CONNECTING*, *OPEN*, *CLOSING* and *CLOSED*.

Events

WebSocket is an event driven protocol, meaning that there are certain events that trigger code to be executed. The API presents us with four events: *open*, *message*, *error* and *close*. The open event is fired as soon as a connection has been established with the server. «Message» is triggered when a new message has arrived. The error event is set off when an error has occurred and the close event is fired when the connection has been terminated.

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

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

Methods

There are two methods you can call on a WebSocket object; *send* and *close*. «send» takes parameter data and sends it over the socket. A WebSocket accepts either String data or binary data, depending on you need. 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. You can optionally provide a status code symbolizing the reason for the closing call and a reason string.

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 have to already 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 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 in the API above is triggered, as 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 X: WebSocket opening handshake example[14]

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 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:

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-------+-+-------------+-------------------------------+
|F|R|R|R| opcode|M| Payload len |    Extended payload length    |
|I|S|S|S|  (4)  |A|     (7)     |             (16/64)           |
|N|V|V|V|       |S|             |   (if payload len==126/127)   |
| |1|2|3|       |K|             |                               |
+-+-+-+-+-------+-+-------------+ - - - - - - - - - - - - - - - +
|     Extended payload length continued, if payload len == 127  |
+ - - - - - - - - - - - - - - - +-------------------------------+
|                               |Masking-key, if MASK set to 1  |
+-------------------------------+-------------------------------+
| Masking-key (continued)       |          Payload Data         |
+-------------------------------- - - - - - - - - - - - - - - - +
:                     Payload Data continued ...                :
+ - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - +
|                     Payload Data continued ...                |
+---------------------------------------------------------------+
```

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

- FIN and RSV bits: The first bit in the frame is the FIN bit. This is set to 0 if there is a following frame to the message being sent. The next three bits are there for protocol extensions. Extensions will be discussed later.

- Opcode: The following four Opcode bits symbol the type of the frame payload. For example if you're sending UTF-8 text this would be 1, while for binary data it would be 2

- Masking: The mask bit indicates whether the payload has been masked or not. All WebSocket messages sent from a client to a server must be masked in order to avoid HTTP proxy issues. If the mask bit is set, a 4 byte masking key is added to the frame header.

- Payload Length: WebSocket frames encode the payload length with a variable number of bits, so that small frames (0-126 bytes) only need 7 bits to encode the length. 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.

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[15], such as Microsoft SOAP and unofficial open protocols such as XMPP. There's of course possible for everyone 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
```

WebSocket Libraries and Deployment

All modern web browsers support WebSocket, but there are sadly still lot a user running out of date browsers and as a result it can be beneficial to a WebSocket emulation library - a library that uses other techniques if WebSocket support isn't there, while keeping a single interface for the programmer to deal with. There are quite a few of those libraries out there, but the two most notable are Socket.IO[16] and SockJS[17]. Both will first try to use WebSockets and fallback to other real-time techniques if browser support isn't there, while keeping the same programming interface. Fallback technology includes Long Polling, HTTP Streaming and even Adobe Flash Sockets for Socket.IO. SockJS tries to emulate the

WebSocket API as close as possible, while Socket.IO adds more features on top. Socket.IO is targeted at Node.js, which is a JavaScript backend platform, while SockJS can be used with other backends. For traditional Java backends like Java EE, WebSocket is also fully supported, with JSR 356[18]. On the .NET side there's SignalR[19] which works the same way.

One would think that these WebSocket emulation libraries would become redundant once all users run up to date browsers with WebSocket support. But that might not be the case. The fact is that the WebSocket API could be too simple and barebones, resulting in a lot of boilerplate code. SocketIO for example gives the abstraction of rooms that users can join and broadcast abilities[20].

WebSockets vs. HTTP

WebSockets are great, but won't replace HTTP. Instead the two protocols will work together to bring real-time web applications to market. There are features of HTTP, that WebSockets don't provide. It doesn't 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.

WebSocket is an easy to use, modern and powerful TCP like protocol, that in some areas even outshines 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. The meta data is there to give a sense of state to an otherwise stateless protocol. Since WebSockets are stateful, message size can be tiny in comparison (but requires more action at the server). With our previous stock price app example each stock price update would not be many bytes sent from the server, probably under 126 bytes, meaning that the total WebSocket frame headers size would be only *3 bytes* (2 for FIN, RSV and Opcode bits, 1 for mask bit and payload length). 3 bytes is a really small overhead compared to what you get with HTTP.

HEAD OF LINE BLOCKING

Server Sent Events

For some real-time applications there are no need for the client-to-server capability that WebSocket provide. The stock price example earlier is a great example of this. All we need is a way to quickly push data from the server to all connected clients. Server Sent Events is an addition to HTML5 that gives us just that. Just like with WebSocket there are two parts of SSE you need to understand. First it is the programmer API called EventSource. Second is the protocol itself.

EventSource API

The EventSource interface is quite small and developer friendly. And is defined like this by W3C:

```
[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 X: The EventSource API[21].

Right off the bat it looks very similar to the WebSocket API, and it is. There are different ready states available and open, message and error events you can listen for. There's also a method to close the connection. 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 X: How to connect to a SSE endpoint and listen for updates.

Event Stream Protocol

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[22].

Syntactically, what differs SSE 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 X: Part of example found in High Performance Browser Networking[22].

In the example above you can see how the server sets the client reconnect interval to 15 seconds. Also server-sent data can be pure text or JSON. Also it’s possible to send an id and custom event associated with that message.

Server Sent Events Adoption

So with a simple API and developer friendly features like automatic reconnects, SSE should be the obvious choice for server push on the web? Sadly that is not the case. The way I see this, 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 would have to be converted using base64 encoding, but this adds some overhead. Second, adoption is not perfect. Every modern browser except Internet Explorer supports SSE, but since IE accounts for a very large portion of the market, choosing SSE alienates those users.

Node.js - JavaScript Everywhere

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[REF], CoffeeScript[REF] and TypeScript[REF], 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 front end JavaScript to an other server side 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;
```

Example X: Blocking code

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

Example X: 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.

Performance testing

Because this thesis will be about benchmarking technologies, it's appropriate to introduce some basics of performance testing. It's become increasingly more important to performance test your system. In these days of the internet and the web, many applications launch to uncertain amounts of popularity. Unexpected high demand lead to load levels that can severely slow down or even break applications.

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's 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 an as efficient solution as possible. Even though the book focused on implementing performance testing into an agile development process for a real world project, it functioned fine as a guide for this thesis. I used to book to learn about what performance testing is, what types of tests can be run and what to look for. Performance testing can be divided into subcategories:

- Load testing: Load testing an application means putting it under a certain amount of demand and measuring it's response time and resource use.
- Stress testing: Stress testing is putting a system under extreme levels of workload and see how far it is possible to push it before it breaks. Stress tests can also help you find the weakest link in your system causing bottlenecks.
- 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.

As stated with the problem statements, the mission of this thesis is to compare HTTP to SSE and WebSocket and try to answer what types of applications that would benefit from WebSocket. To get an accurate picture, I've chosen to load and stress test the technologies in question. In my view, the most interesting results of the tests are:

- Server CPU load and memory usage for the load tests.
- Server CPU load, memory usage and message throughput for the stress tests.

With message throughput, I mean the number of messages the server can handle per second.

Project Part 1

Server-To-Client

Introduction

For both the stock price app and chat app examples presented in the background piece, the ability to push data from server to connected clients is essential. Consequently, this part of the thesis concerns the server-push capabilities of the Web. I've decided to compare the common real-time pattern HTTP Long Polling to the newer Server Sent Events and WebSocket and see who comes out on top in terms of server performance.

This project part is divided into the following two parts:

- Methodology - The test scenario and all choices made in regards to it are presented here. There are also some notes on the development of the tests.
- Results - Graphs and textual description of the test results are presented here.

Methodology

Test scenario

The 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 using either WS, SSE or HTTP and the server connects to the backend system using a persistent connection. The backend regularly sends messages to the server and it's 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. The clients are then real web browsers connected to a web server that in real time, based on updates from a stock broker, updates the prices of several different stocks.

Figure X: Simple diagram showing the three components

Monitoring

What To Look For

With this being a performance test of WS, SSE and HTTP Long Polling, they must be compared on some common grounds. I've decided to look into two metrics on the server side - *CPU load* and *memory usage*, as well as one from the client viewpoint - *response time*. This way there will be clear to see when the different protocols meet their limits.

How To Get It

To obtain the values needed, the are two monitoring components, one monitoring process running on the server machine and one ping client running separate from the normal client processes. How these are created and how they report the results are explained below.

Variables and Constants

The Number of Connected Clients

This is the most interesting metric and the only variable. The tests will be run all the way from only one connected client up to the amount that makes the response time really start to climb.

The Number of Messages the Backend Should Send

This number is not particularly important as long as it's high enough for the tests to run long enough that the monitoring process will get an accurate picture. Consequently it is a constant set to 2000.

How Long the Backend Should Wait In Between Messages

This is a constant and set to 5 milliseconds. This way the tests will not run so long that it takes unreasonable time to do the testing. At the same time this should increase CPU load on the server so that the the break point for the response time should happen within a manageable number of clients.

The Size of each Message

This is the last constant. It should resemble a real world message size, so I set it to just shy of 1KB - 924 bytes.

Detailed Information Flow

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. If the user on the server presses the return key, the go message is sent. It's up to the user on the server to make sure all clients are connected before sending the go message to the server.

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 a monitoring process that right before and during broadcast monitors the CPU usage and memory footprint of the server 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 of the broadcast end.

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

There's a "master" client process that has two jobs:
- Fork up a given number of client processes that connects to the server.
- Fork up a ping client.

The client processes immediately connects to the server and reports to the master client when they are connected. This way the master client can tell when all clients have obtained connection. 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 single process that every 50th milliseconds sends a message with a timestamps to the server. The server instantly "pongs" 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.

Figure X: Sequence diagram showing detailed information flow

Hardware Setup

As the point of this thesis part is to load and stress the the server, it is critical to isolate the server from the clients and backend. This can be done in different ways:

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 ideal. It's 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's 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's 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.

It is important that the machine the server ran on is considerably slower than the one with the clients and the backend, since the server must reach its resource limit before the client machine. 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 programming environment) 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 setup:

2013 MacBook Air
Dual Core Intel Core i5 1.3 GHz
8 GB DDR3
Mac OS X 10.9.2

The backend and clients ran on this machine:

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

As I didn't want the network to be a bottleneck, I decided to have them both running on a cabled 1 Gb/s network. Since none of the machines actually have ethernet interfaces built in, a Thunderbolt-to-Ethernet dongle was used[REF].

Programming environment

Node.js was not presented in the background part for no reason, it is the chosen development platform for the practical part of this thesis. The reasoning behind this choice:

Node.js is lightweight, fast and scales well

For this thesis I wanted to benchmark protocols against each other, not web frameworks. Node.js provides very little overhead and since it scales great, the scaling part of this thesis will not be hampered by blocking server code.

Node.js is single threaded

The fact that Node.js only uses one OS process, makes it perfect to monitor without interfering with the server process, as long as the server has more than one CPU core.

Node.js is a platform with cutting edge innovation

When looking at GitHub's most trending and popular repositories[REF], Node.js is the web framework that by far has the most packages and traction. One reason for this is its user friendly package manager NPM[REF].

I only write code in JavaScript

As a result of choosing Node.js for the work in this thesis, all code is written in JavaScript. There are a few reasons why I think this is a good thing:

- JavaScript is a very 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's even a JavaScript interface to the OS[REF]. Also, I chose it so that I can learn its quirks[26], 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 [REF]. This makes it perfect for the clients.

Command Line Clients

Kristian Johannessen[REF] had several challenges with his tests and suggested using lightweight console apps or headless browsers instead of web browsers for future work. 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 they consume quite a lot of system resources. Self-written console apps gives full control and let me put my focus on what I want - the transports.

TODO: Søk etter PayPal - Node.js

Regarding Libraries

The point of this thesis was to test and benchmark different transports - protocols. Benchmarking protocols doesn't really make sense as there are several implementations of different protocols. Testing certain implementations are, on the other hand, of course possible. It was clear from the start that I didn't wanted the thesis to be about comparing different libraries or frameworks, Kristian Johannessen[REF] already did that.

As Node.js is just a simple JavaScript runtime and not a full-blown Web Framework, I had to rely on some libraries. The libraries had to be as small and bare-bones as possible to lay 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 master thesis on its own[REF]. 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[REF] 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[REF] WebSocket implementations for Node.js, making it perfect for testing. In fact, since ws is small and fast, it serves as the low level WebSocket implementation for Socket.IO.

SSE

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[REF]. 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. Details on the implementation are below.

HTTP Long Polling

I needed at two routes into the server, one for normal clients and one for the ping client. Therefore, I chose to use the very popular web framework Express[REF].

Resource monitoring

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

Notes on Development

Backend

The backend system is essentially a WebSocket server using the same library, ws, as the server component. Whenever a client (broadcast server in the test sense) connects, the backend requests broadcast information and awaits a go signal. When all messages have been sent, the backend sends a final done message indicating that the test is over. The backend doesn't close the connection to the server, that responsibility is left to the server itself.

WebSocket

As WebSocket is a persistent connection protocol, each message received from the backend is immediately distributed to connected clients and never stored locally.

The WebSocket clients are started by the program startwsclients.js and given the server address and port over Node.js' interprocess communication method process.send[REF]. The clients discard each broadcast message as they arrive and just keeps track of how many it has received. The server closes the connection and the client report to it's mother process if all messages have arrived.

SSE

As with WebSocket, each SSE client has a persistent connection to the server during the broadcast phase. This way each backend message can be discarded after broadcast. As mentioned earlier, the server implementation of SSE was self-written:

```
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 X: From sseserver.js - the SSE endpoint.

To conform to the SSE specification, the timeout is set to infinity and Content-Type to text/event-stream. This is essentially all needed for a HTTP server to become SSE-ready.

The clients are started by the program startsseclients.js, uses the EventSource library and works the same way as the WebSocket clients.

HTTP Long Polling

Consider the example in figure XXX in the background part. This shows the server need to store each message it receives from the backend. In addition, when requesting new messages, each client must tell the server how many messages it has received, so that the server knows what message or messages to send back, essentially sequence numbers. This introduces a level of complexity not seen in the WebSocket and SSE servers. It can be expected that server CPU usage here is quite a lot higher than for the other servers.

The clients started by starthttpclients.js are also similar to the WebSocket and SSE ones, except that they naturally need to send a new request after each message received.

Resource Monitoring

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.

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.

How To Run the Tests

First, Node.js must be installed to run the tests. Second you must navigate to the right folder before running the commands below. The backend must be started before the servers and the servers must be started before the clients.

The backend will always listen for connections at port 9000, and the three different broadcast servers will listen on port 8000.

Before starting the broadcast, wait for all clients to connect to the server.

Backend

```
$ node backend.js
```

WebSocket Server

```
$ node wsserver.js <backend ip> <backend port>
```

Example:

```
$ node wsserver.js localhost 9000
```

SSE Server

```
$ node sseserver.js <backend ip> <backend port>
```

Example:

```
$ node sseserver.js localhost 9000
```

HTTP Server

```
$ node httpserver.js <backend ip> <backend port>
```

Example:

```
$ node httpserver.js localhost 9000
```

WebSocket Clients

```
$ node startwsclients.js <server ip> <server port> <number of clients>
```

Example:

```
$ node startwsclients.js localhost 8000 128
```

SSE Clients

```
$ node startsseclients.js <server ip> <server port> <number of clients>
```

Example:

```
$ node startsseclients.js localhost 8000 128
```

HTTP Clients

```
$ node starthttpclients.js <server ip> <server port> <number of clients>
```

Example:

```
$ node starthttpclients.js localhost 8000 128
```

Results

HTTP Long Polling

CPU Load

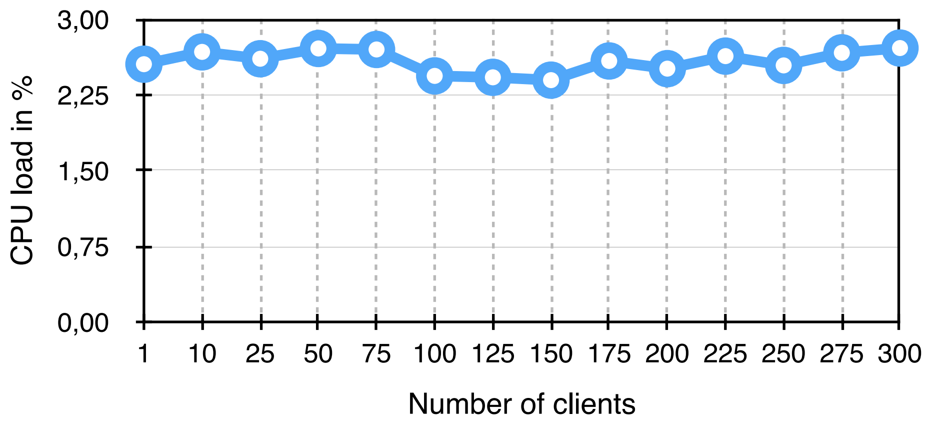


Figure X: Average CPU load before broadcast

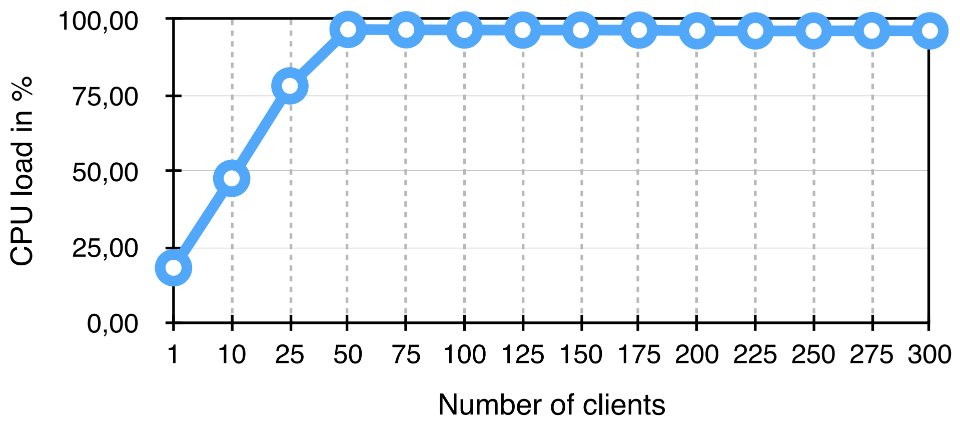


Figure X: Average CPU load during broadcast

For the CPU load before broadcast, there are no sign of increased CPU load as long as the clients are idle. The CPU usage is stable below 3 % all the way up to 300 clients.

During the broadcast on the other hand, the CPU load escalates very quickly and at 50 connected clients it seems to utilize the entire delegated CPU core.

Memory Footprint

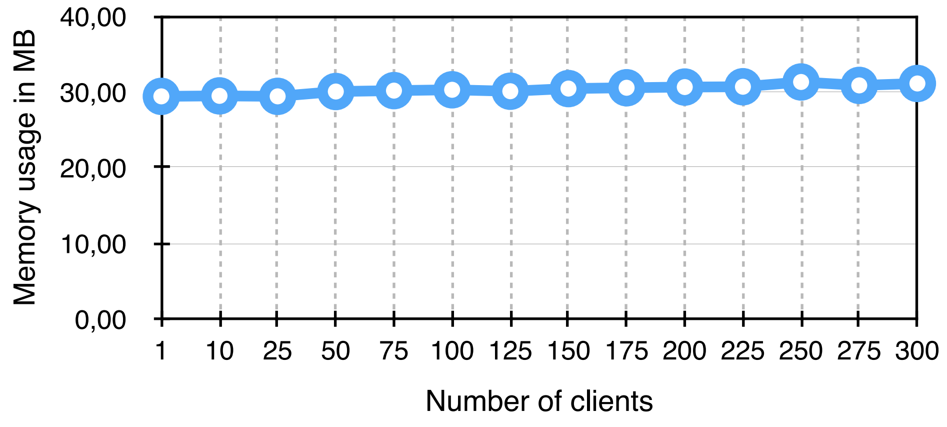


Figure X: Average memory footprint before broadcast

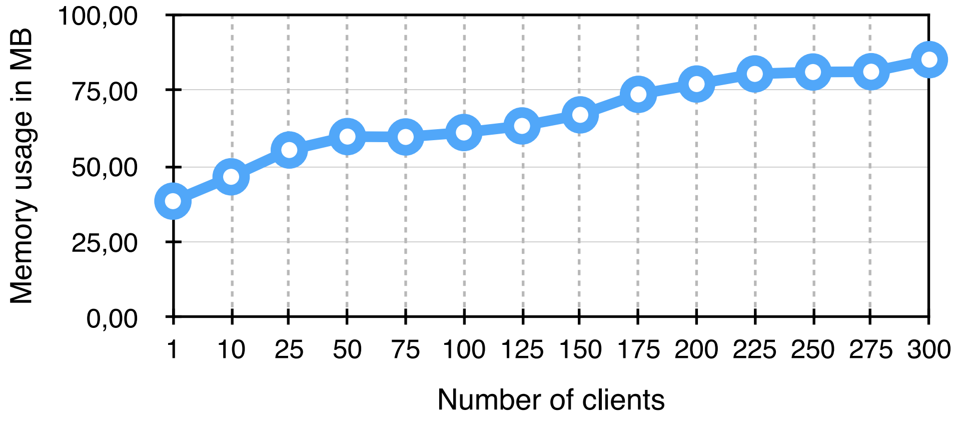


Figure X: Average memory footprint during broadcast

As with the CPU load before broadcast, the memory footprint also stays roughly the same for all client numbers.

During broadcast we can see a steady climb up to 85 MB when 300 clients are connected.

Response Time

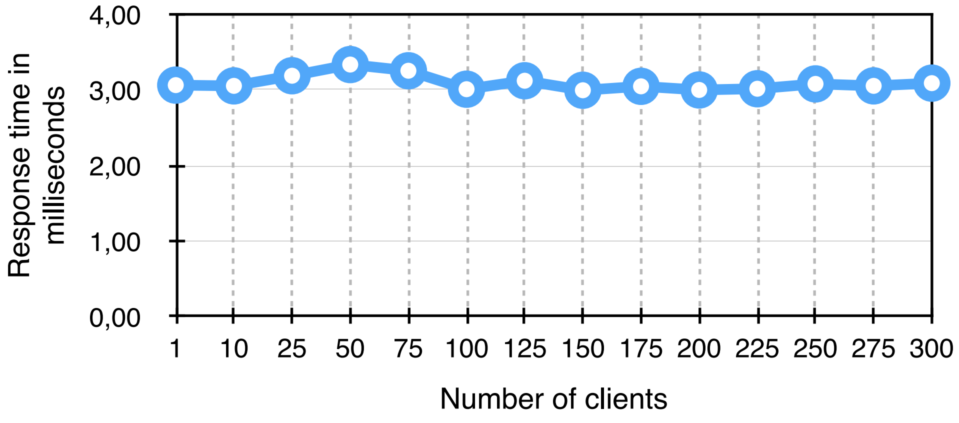


Figure X: Average response time before broadcast

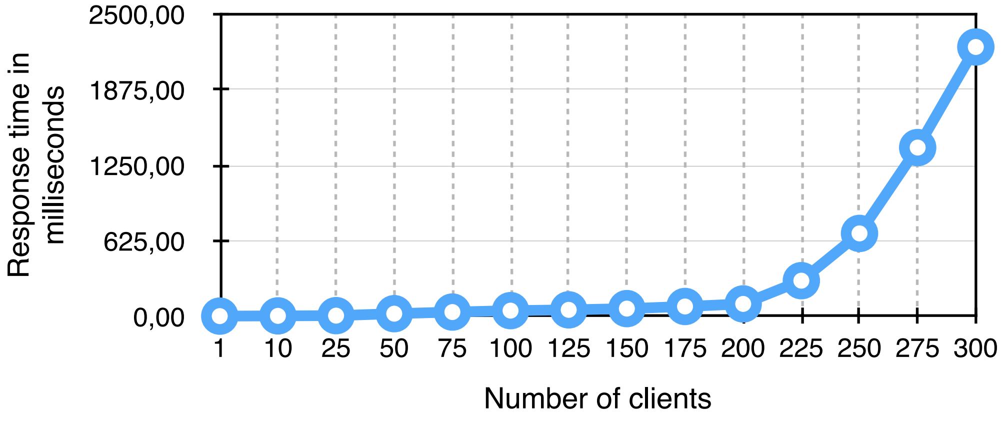


Figure X: Average response time during broadcast

Once again before broadcast, the numbers don't seem to be affected by the number of idle clients. It stays comfortably below 4 milliseconds the entire time.

During broadcast we see a steady climb all the way from the start, but from 200 to 225 clients there's more than a doubling of response time. At 300 clients the response time is as long as whole 2,2 seconds.

Server Sent Events

CPU Load

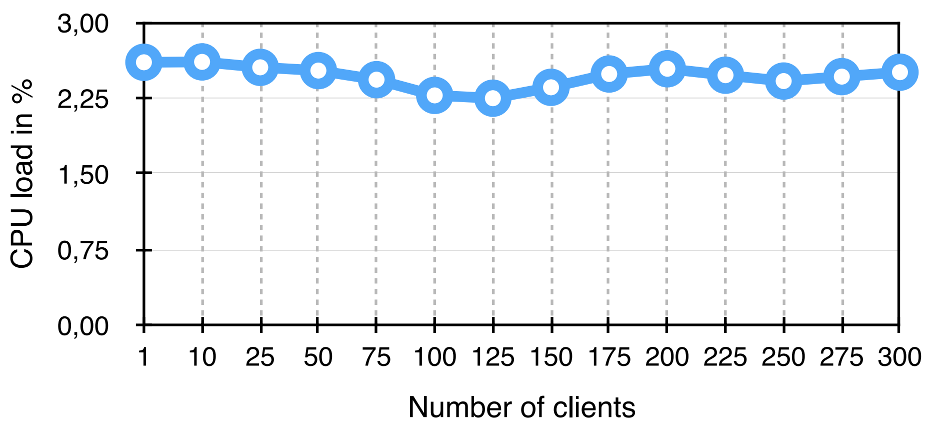


Figure X: Average CPU load before broadcast

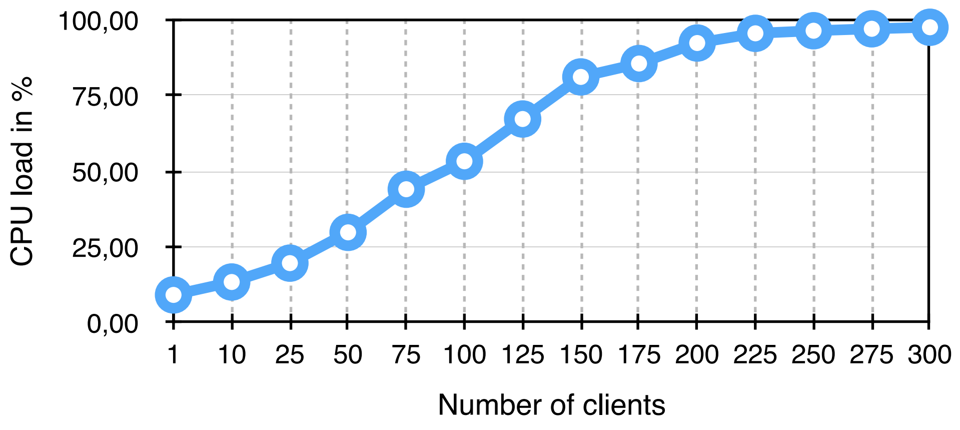


Figure X: Average CPU load during broadcast

Once again, the CPU load is consistently low for all numbers of idle clients.

During load we see a steady increase reaching peak at around 225 to 250 clients.

Memory Footprint

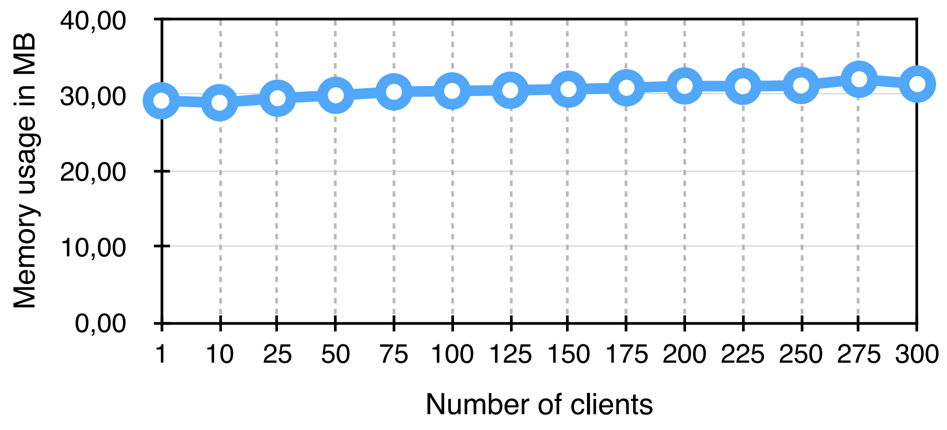


Figure X: Average memory footprint before broadcast

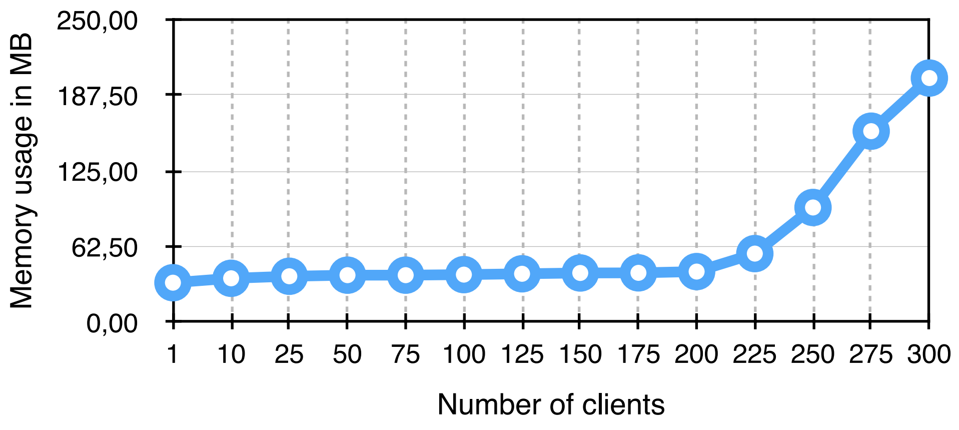


Figure X: Average memory footprint during broadcast

The memory footprint is fairly low and stable although increasing some with the number of idle clients.

During load on the other hand, the memory footprint is rapidly building up from 225 to 250 clients.

Response Time

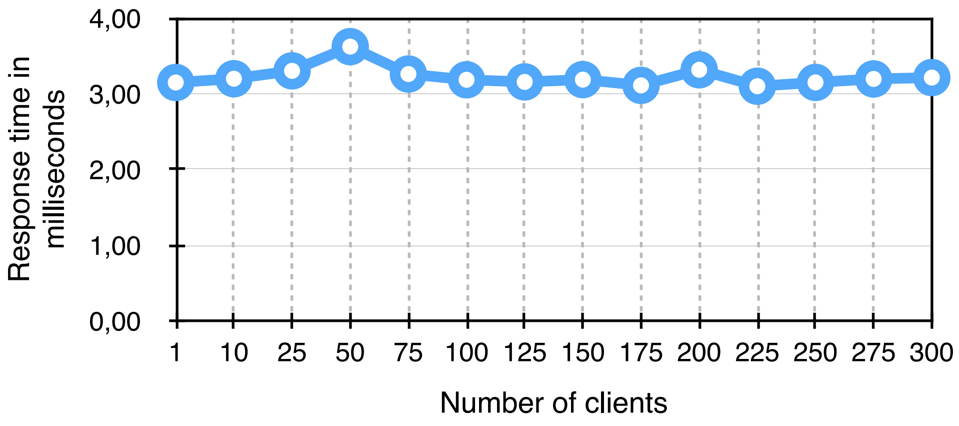


Figure X: Average response time before broadcast

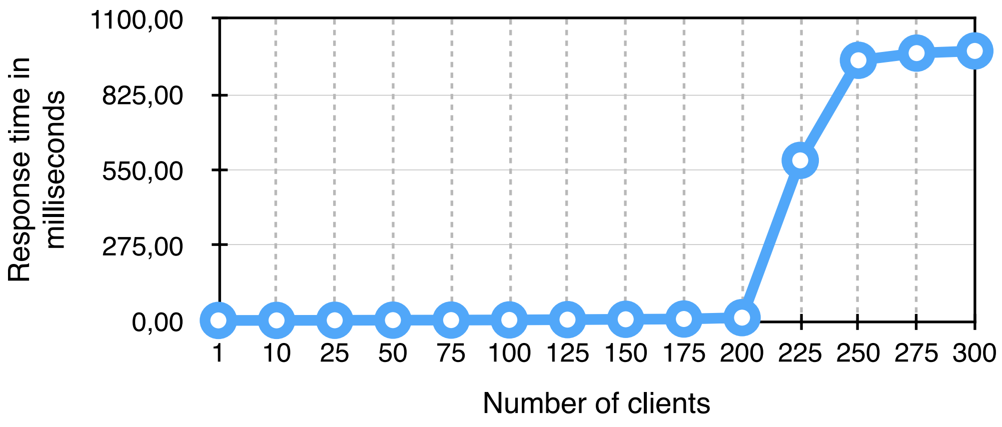


Figure X: Average response time before broadcast

Response time for the ping client stays low and stable for all numbers of idle clients.

During load we can see a clear break point for the response time starting at 225 clients. The response time then somewhat plateaus at about 1 second.

WebSocket

CPU Load

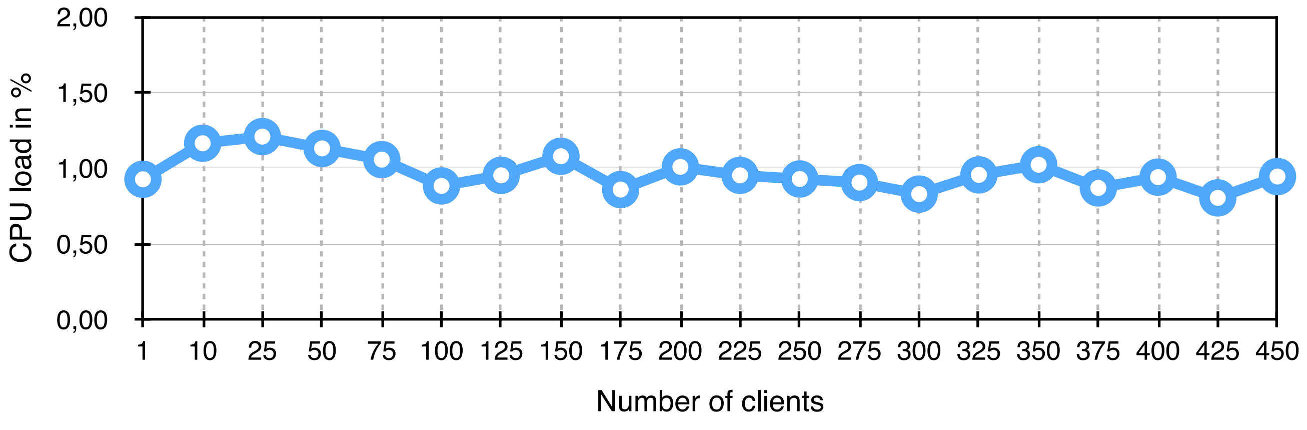


Figure X: Average CPU load before broadcast

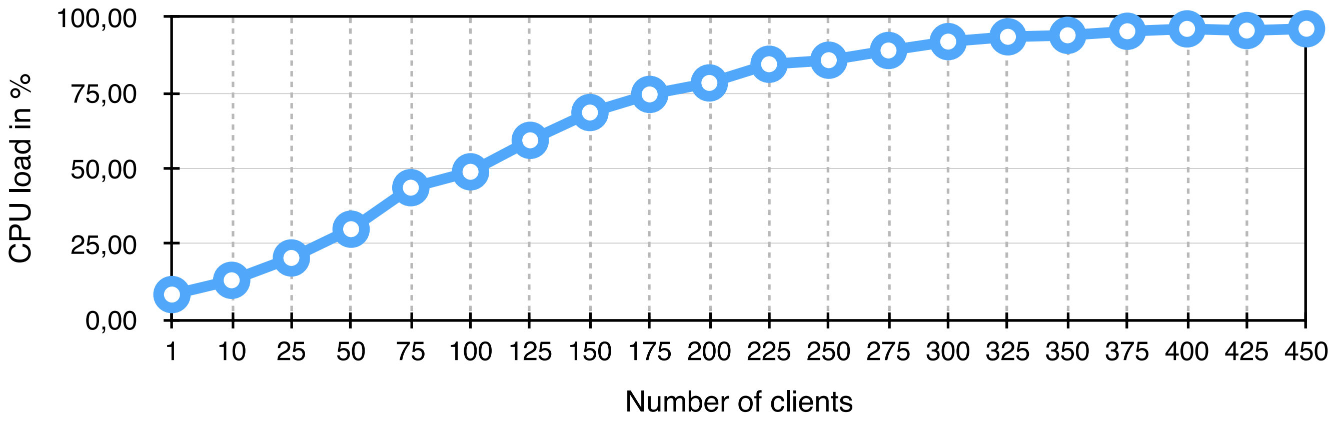


Figure X: Average CPU load during broadcast

As with HTTP and SSE, the idle CPU load is low and stable for WebSocket as well.

During load, the CPU load slowly and steadily gets larger and reaches peak at around 375 to 400 clients although it is above 90% from 300 clients.

Memory Footprint

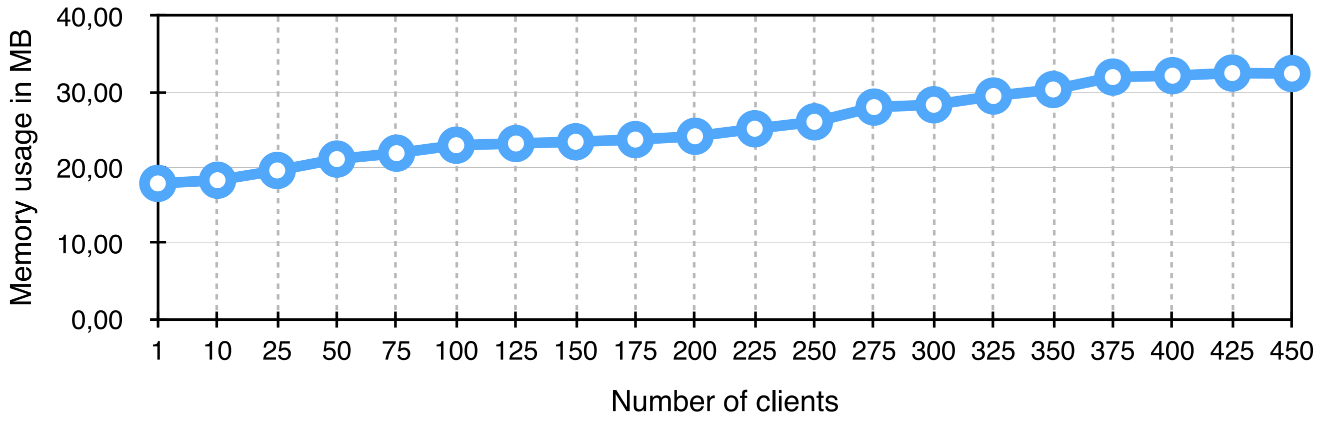


Figure X: Average memory footprint before broadcast

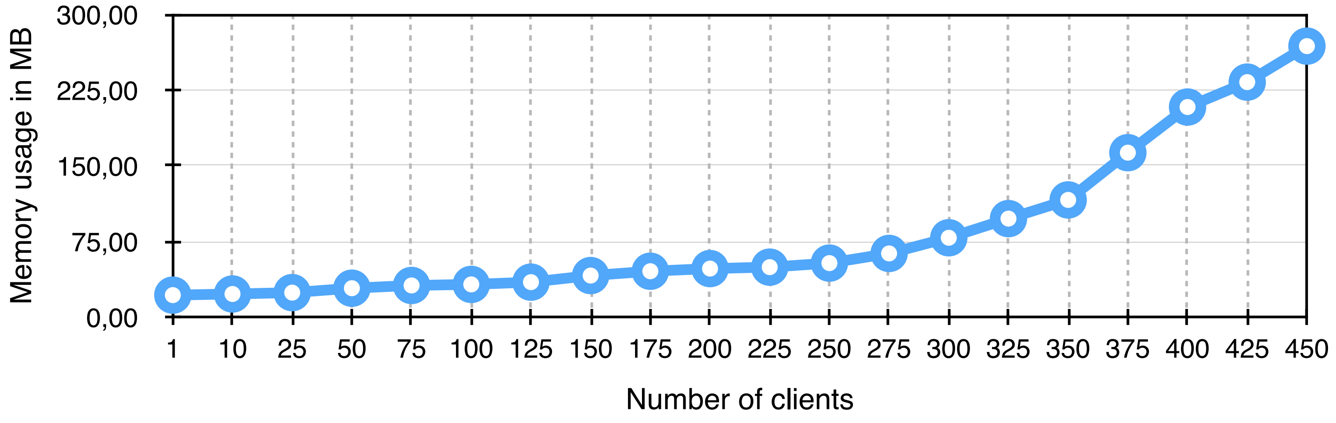


Figure X: Average memory footprint during broadcast

Idle memory use consistently increases with the number of connected clients.

During load, we can as with SSE, see a swift increase from around 350 clients and up.

Response Time

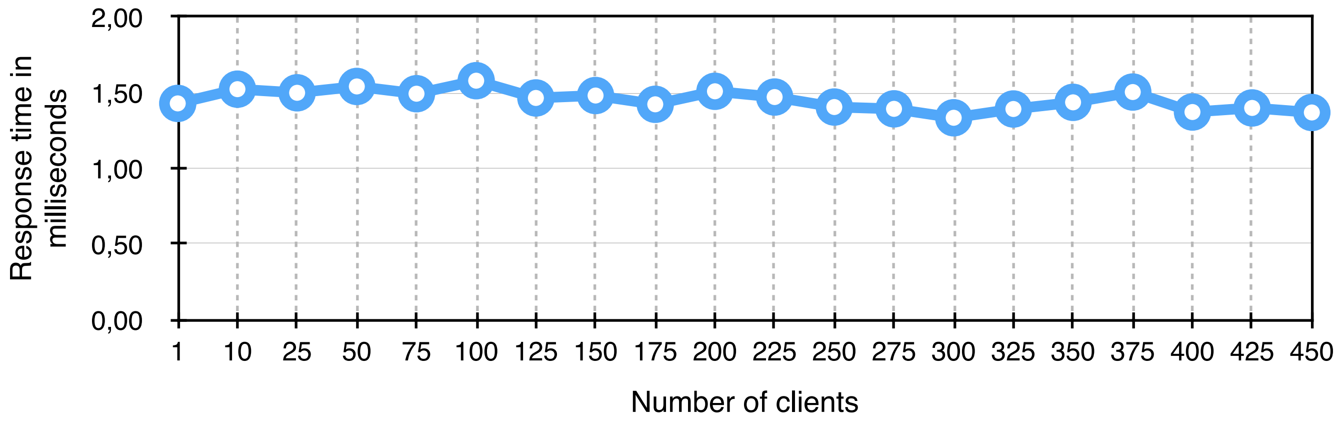


Figure X: Average response time before broadcast

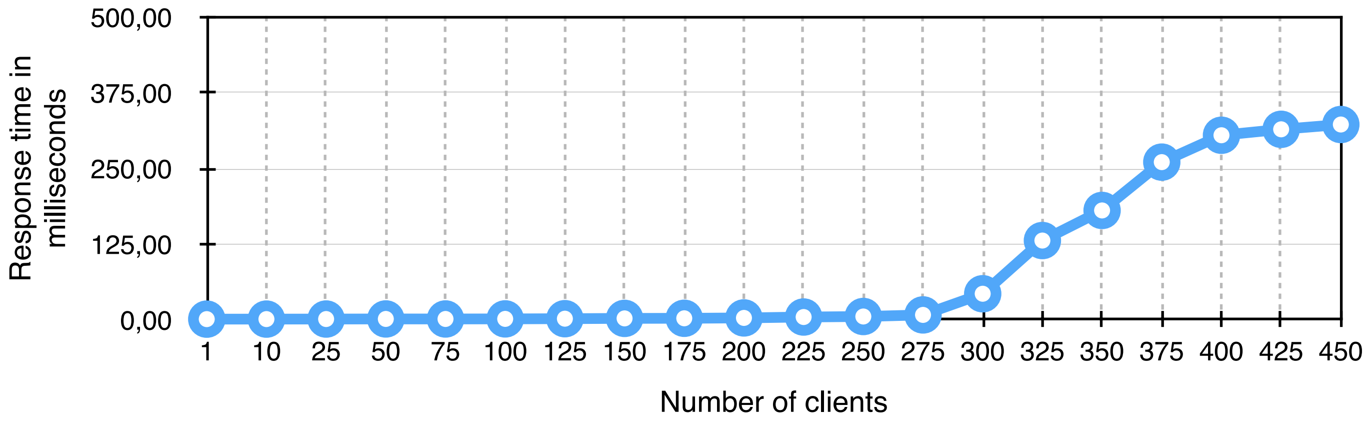


Figure X: Average response time during broadcast

The average response times for idle clients are incredible low just above 1 millisecond all the way up to 450 clients.

During load on the other hand, response time stays low to around 275 clients. After that we see a climb that to a certain degree plateaus from around 400 clients.

Combined

CPU Load

● WebSocket ● Server Sent Events ● HTTP Long Polling

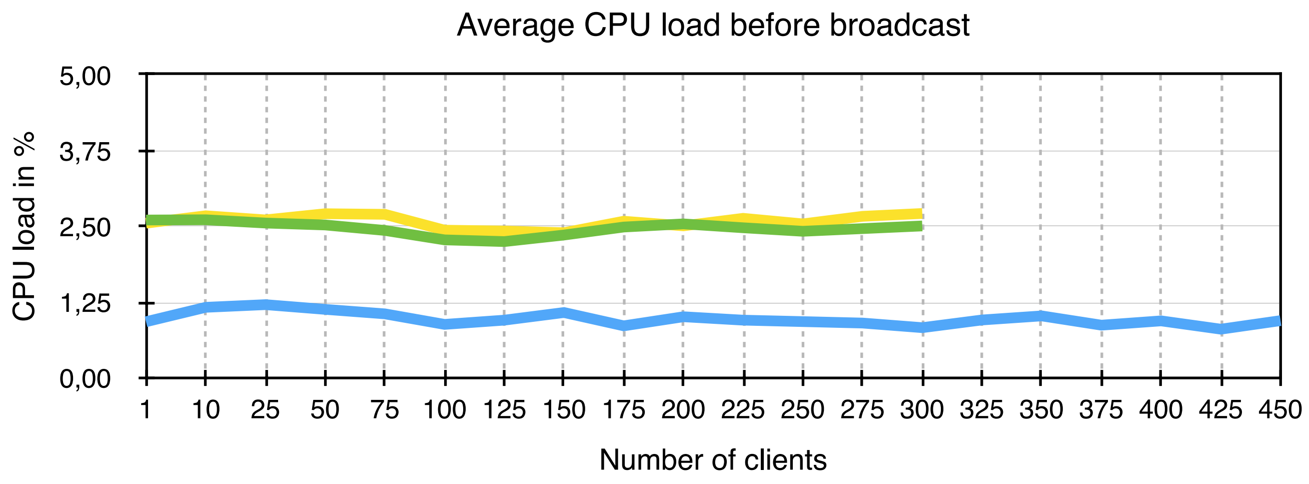


Figure X: Combined chart showing average CPU load before broadcast

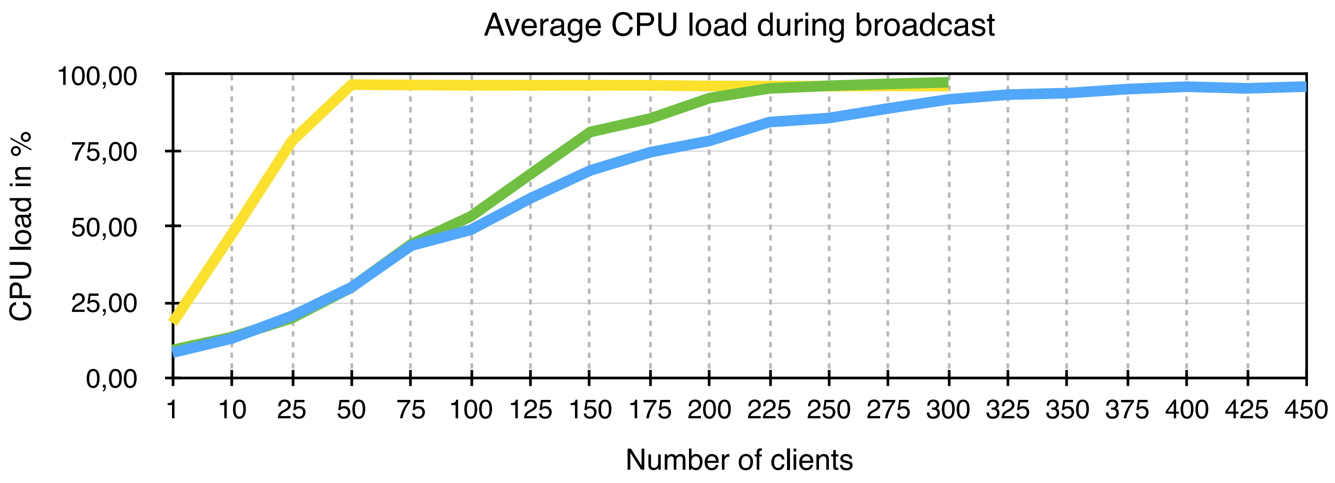


Figure X: Combined chart showing average CPU load during broadcast

With the idle clients, SSE and HTTP performs almost identical. WebSocket on the other hand performs better.

During load WebSocket requires less CPU load for the same job, although SSE outperforms HTTP especially at low number of clients, before converging around 225 clients.

Memory Footprint

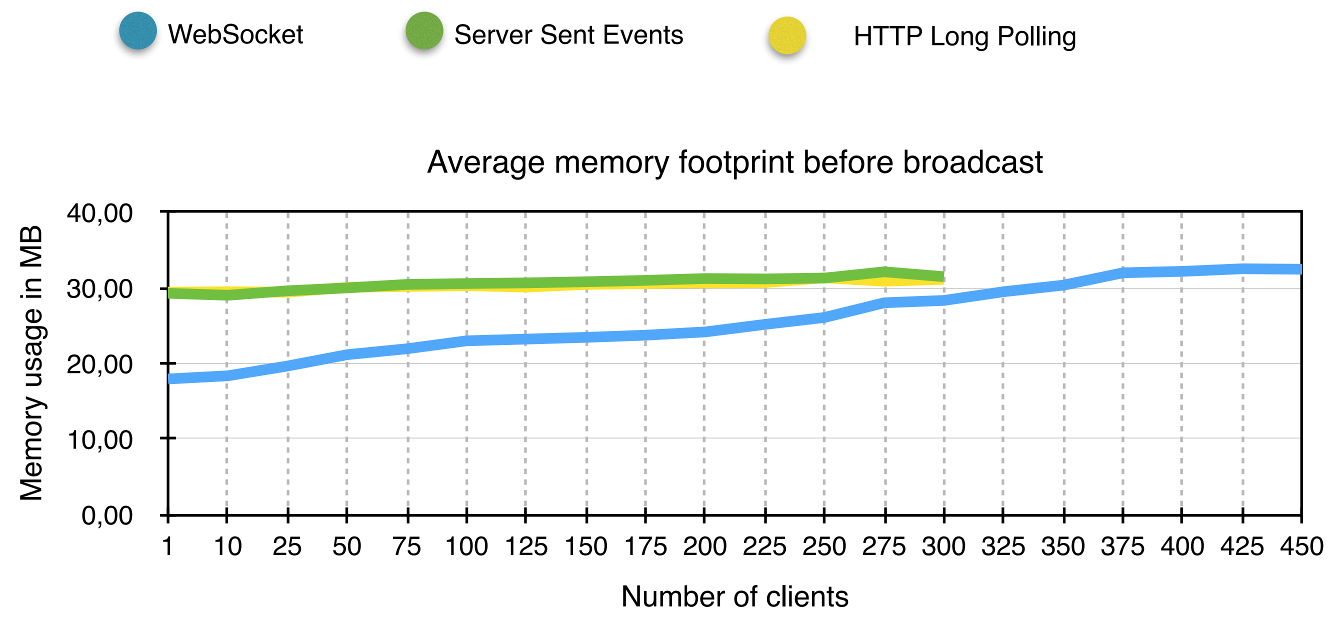


Figure X: Combined chart showing average memory footprint before broadcast

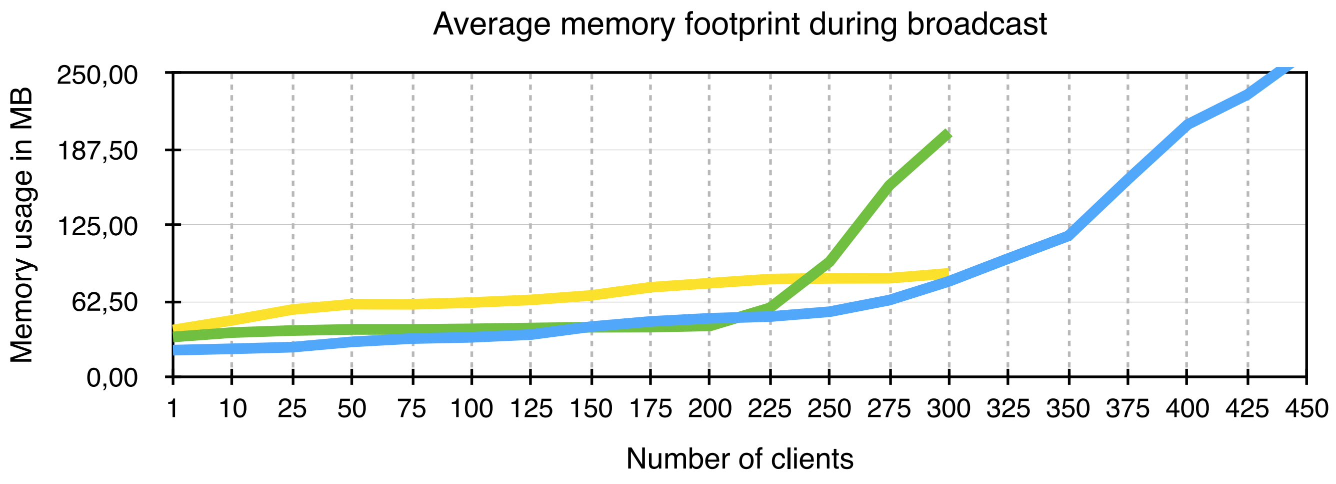


Figure X: Combined chart showing average memory footprint during broadcast

At first, the HTTP and SSE implementations use about the same amount of memory for idle clients, with WebSocket using a bit less. As the number of clients rises, we can see that WebSocket reaches the same level.

The memory footprint during load is more interesting, with both SSE and WebSocket skyrocketing at some point, while HTTP staying stable and low.

Response Time

● WebSocket ● Server Sent Events ● HTTP Long Polling

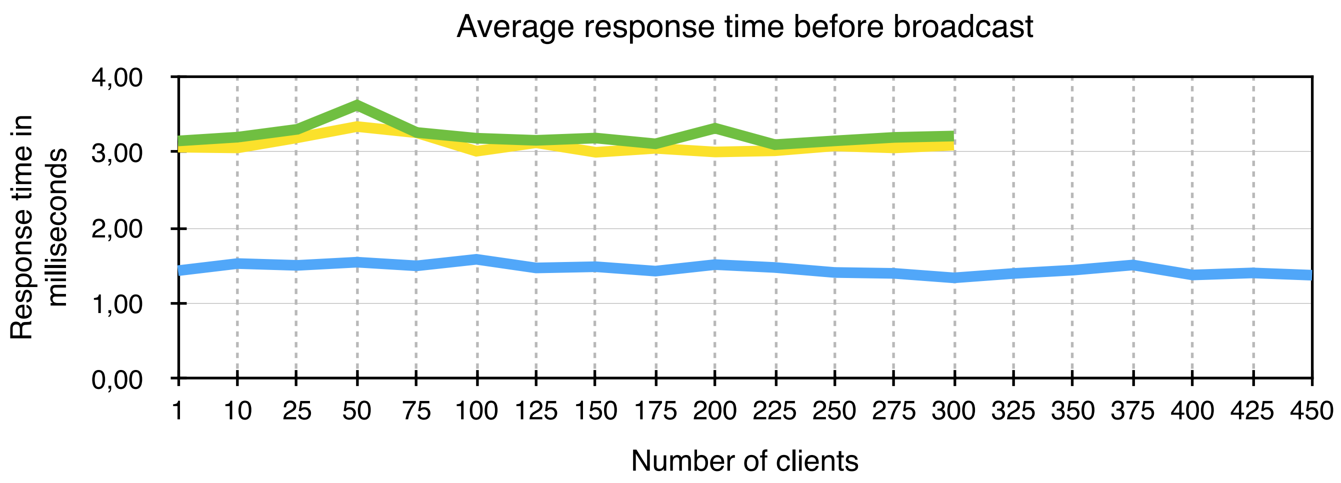


Figure X: Combined chart showing average response time before broadcast

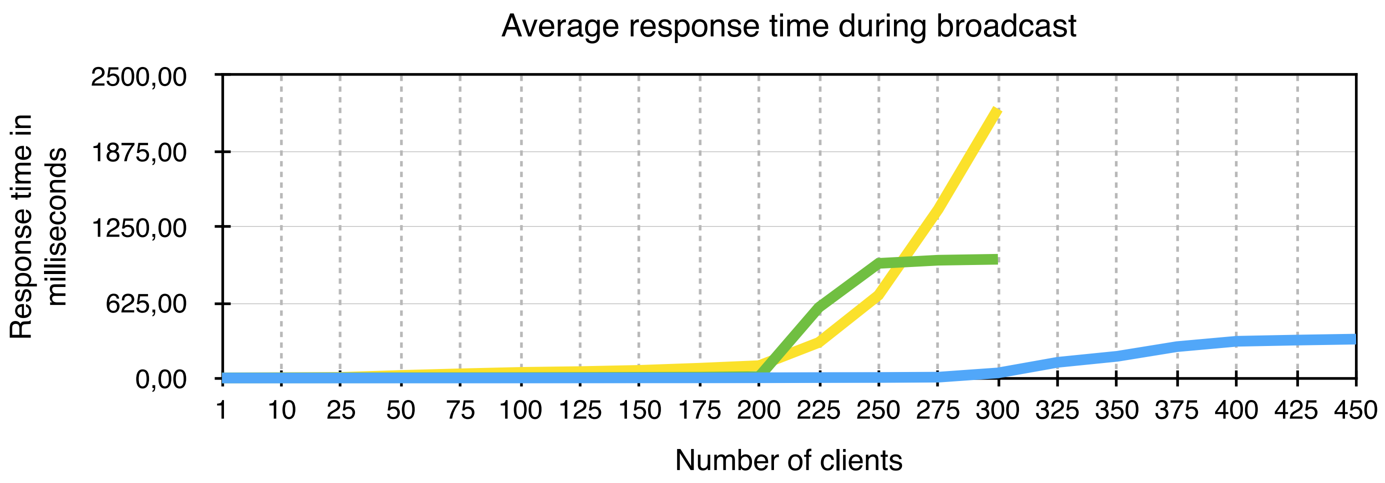


Figure X: Combined chart showing average response time during broadcast

As both the SSE and the Long Polling implementations use the same ping client over HTTP, the response time is almost identical, staying low and stable. WebSocket is even better.

Under load, we can see both HTTP's and SSE's response time increase about the same time, with HTTP plateauing around 1 second and SSE skyrocketing. WebSocket on the contrary performs exceptionally well.

Project Part 2

Client-To-Server

Methodology

Text

Test environment

Text

Results

Text

Analysis

Analysis heading 1

Conclusion

Text

Further work

Text

Appendix

Text

Bibliography

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

2. Connolly, D. A Little History of the World Wide Web. 2000; Available from: http://www.w3.org/History.html.

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

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

5. W3C. Open Web Platform Milestone Achieved with HTML5 Recommendation. 2014; Available from: http://www.w3.org/2014/10/html5-rec.html.en.

6. Adobe. PhoneGap. 2014; Available from: http://phonegap.com.

7. Apache. Apache Cordova. 2014; Available from: https://cordova.apache.org.

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. Engin Bozdag, A.M.a.A.v.D., A Comparison of Push and Pull Techniques for AJAX. 2007.

12. Vanessa Wang, F.S., Peter Moskovits, The Definitive Guide to HTML5 WebSocket. 2013: Apress.

13. I. Fette, A.M. RFC 6455. 2011; Available from: https://tools.ietf.org/html/rfc6455.

14. WebSocket protocol handshake. 2014; Available from: http://en.wikipedia.org/wiki/WebSocket#WebSocket_protocol_handshake.

15. IANA. WebSocket Protocol Registries. 2014; Available from: https://www.iana.org/assignments/websocket/websocket.xml.

16. Rauch, G. Socket.IO: the cross-browser WebSocket for realtime apps. 2012; Available from: http://socket.io.

17. SockJS. 2014; Available from: https://github.com/sockjs/sockjs-client.

18. Oracle. JSR 356. 2014; Available from: https://jcp.org/en/jsr/detail?id=356.

19. SignalR. ASP.NET SignalR. Available from: http://signalr.net.

20. Rauch, G. Socket#in(room:String):Socket. 2014; Available from: https://github.com/learnboost/socket.io/#socketinroomstringsocket.

21. W3C. Server-Sent Events. 2009; Available from: http://www.w3.org/TR/2009/WD-eventsource-20091029/.

22. Grigorik, I., High Performance Browser Networking. 2013: O'Reilly.

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