Real Time Web Applications
Comparing frameworks and transport mechanisms

Kristian Johannessen
Master's thesis
University of Oslo
Department of Informatics
kjohann@ifi.uio.no / kris-90@live.no

May 1st 2014

Abstract

In a real time web application, clients (browsers) receive updates from a server as soon as they occur. With HTML5, a new protocol for this purpose was born: WebSockets. This thesis compares this protocol to different HTTP methods for achieving real time in its second part. The first part compares five different frameworks for real time web applications from a programmer's perspective. Through this, I seek to find the most usable framework for real time, the one with the best performance and to see how WebSockets is better than HTTP for this purpose.

Contents

List of figures

List of examples

List of tables

Preface

This thesis is part of my Master's degree in *Informatics: Programming and Networks* at the Department of Informatics, University of Oslo. At the University, I have received guidance from the research group "Object-orientation, modeling, and languages" (OMS).

I approached the Norwegian company BEKK Consulting with the topic of this thesis, and they let me write about it on a contract with them. The main part of the work has been overseen by one of their senior developers.

All opinions made in the thesis reflect the author's opinions only. They are not representative for those of the University of Oslo nor BEKK Consulting.

I have received some help from the Italian company Weswit regarding server configuration for a framework called "Lightstreamer". They also provided a free license for the product. Otherwise, they have not affected my work or influenced my opinions.

Acknowledgements

First of all I want to thank my supervisor at BEKK, Torstein Nicolaysen. I also like to thank my supervisor at the University of Oslo, Dag Langmyhr. Both have provided support and valuable insight throughout the process.

I also want to thank Weswit for the help with licensing as well as server configuration.

Finally, a great thanks to Team Rubberduck. Without your support, this work would have been much more lonely. You guys are the best!

1   Introduction

In a real time web application, clients (browsers) receive updates from a server as soon as they occur. Google Docs is a typical example of a real time web application. Here, users can edit a document at the same time and simultaneously see what the others do in real time.

The concept of real time web has existed for quite some time. But over the last few years it has begun to gain popularity. With the introduction of WebSockets, we have gotten a new protocol designed entirely for this purpose. I will give an introduction to the motivations behind this protocol and the concept of real time in the background part.

Real time features in web applications have become more common. Several frameworks have surfaced to aid the development of such features. This thesis will look at some of these frameworks. I have developed an application with real time features to test the usability of five frameworks. With usability I mean from a programmer's perspective.

Getting some hands on experience with different frameworks also helped determine if they are needed at all. A real time feature can range from a simple task to a dominating aspect of a web application. This thesis discusses if you need a framework no matter the task.

This thesis will also look at various performance aspects. First of all the performance of different frameworks using different methods for real time communication. But I will also look at how these methods compare to each other. This is the most interesting aspect in my opinion, as it can give insight into the importance of the WebSocket protocol.

It is a clear separation of themes in this thesis. On one hand you have the usability aspects of various frameworks. On the other you have the performance of these and of methods for real time. To handle this, the project work is separated into two parts. The first part covers the development aspect, while the other handles performance.

Both parts are summarized in a common conclusion part. This allows me to make a total evaluation of the frameworks. It also allows me to see trends in the transport mechanisms' performance with the different frameworks.

1.1 Problem statement

The problem statements in this thesis are as follows:

I: How are the frameworks that are featured in this thesis with respect to usability?

Usability is a nuanced property of a framework. Some people may regard certain aspects, like programming environment, important, while others are indifferent towards it. In this thesis, I will evaluate the usability of each framework based on my opinions. However, I will strive to do so within objective boundaries, discussing each usability property thoroughly.

II: Which of the frameworks have the best performance in terms of message frequency, latency, network usage and server resource consumption?

Sometimes, performance is an alpha omega requirement for a project. Even if it isn't, one will want a framework with good performance most of the time. The performance aspects measured in this thesis are the ones I deem most important for real time web applications.

III: Are there any real time web applications that may benefit from not using the aid a framework provides?

If the functionality you need is simple, does your project benefit from an extra dependency? Or can you make it yourself without much extra work? Sometimes you would like as few dependencies as possible. But if it takes up a lot of time to do something yourself, a framework may be a better solution.

IV: Does WebSockets outperform the old, established HTTP methods for real time in terms of network usage, message latency and use of machine resources?

If it turns out that WebSockets doesn't perform better, is it something we need?

V: Can WebSockets be the foundation for the next generation of HTTP?

The answer to this is coupled with the previous question. If WebSockets is just as good as HTTP performancewise, it will not be the foundation for the next generation of HTTP. However, if it is better, can it change the way browsers communicate with servers in more aspects than real time?

1.2 Related work

I have not been able to find any studies that compare frameworks for real time web applications on the same scale as this thesis. Some comparisons I have found of frameworks for real time, are long lists of names and some features [1], [2]. They do not provide any sort of in depth study, just some bullet points for each.

There are some more thorough comparisons, but these focus on only two frameworks. Some focus on usability [3], [4], while others offer a comparison of performance [5]. I do not find all of these reliable, though. Some comparisons are publications on a certain framework's homepage, comparing their own product to someone else's [6], [7].

When it comes to comparing the different transport mechanisms for real time, most are older papers. A paper by Bozdag et al. [8] is an example of such. It compares AJAX approaches to real time (WebSockets didn't exist when it was written). It concludes that HTTP-streaming gives the best data coherence. In addition, it points to some performance challenges with this approach. The paper also mentions a lot of related work. All these papers are, of course, even older, so WebSockets is not featured in either of them.

Jõhvik [9] wrote a bachelor's thesis in 2011 that seems to be based on the work of Bozdag et al[1]. At this time, WebSockets did exist, but he dismissed using it "due to browser incompatibility" [9]. His conclusion contradicts the one from the paper he builds on, stating that long-polling was best with his setup.

Few research articles feature a comparison of WebSockets and HTTP methods for real time. A study conducted by Darshan et al. supports this claim [10]. Their work remains the only research article I have found that does anything that resembles my work. They focus on network traffic, and conclude that WebSockets uses 50% less than HTTP-streaming. Furthermore, it concludes that WebSockets clients are a lot more efficient than HTTP clients. Their study shows that a WebSockets client can send up to 215% more data using the same bandwidth as a HTTP client.

This thesis focus a lot on frameworks. The load tests in the second part of the project use the context of each framework to compare transport techniques. As far as I can tell, no other study has done this. All tend use one manual implementation for each transport.

1.3 Terminology

This section describes certain terms and phrases used throughout the thesis.

I use the terms "library" and "framework" to describe what I am testing in this thesis. A library is a much smaller entity than a framework [11]. For the sake of consistency, I use framework as a general term. When referring to a library in specific, I use that term.

The term "app" in the context of this thesis, refers to a real time web application. Not a mobile app.

---

[1] The first source of the thesis is Bozdag et al. and Jõhvik's approach is very similar.

The term "transports" refer to various mechanisms for real time communication. These include WebSockets, Server-Sent Events, HTTP-streaming, long-polling and polling.

A server can refer to either a physical machine or a piece of software. In this thesis, it means the latter.

"WebSockets" appears as a plural word in this thesis, but it is not treated that way. It refers to the WebSocket protocol, and is treated just like the term "HTTP". Thus formulations like "WebSockets is..." and "HTTP is..." appear in this thesis.

Automated tests are mentioned several times in this thesis. Such tests involve several types of tests, but unit testing, integration testing and functional testing are most relevant to this thesis. Unit testing refers to testing a single unit [12], or class in an object oriented language. Integration tests puts several units together [12]. In this thesis, I use the term about testing the data access layer of applications[2]. Functional tests, test behavior of an application as a whole [12]. In this thesis, this kind of testing is done using browsers.

I use "Node" as a short term for Node.js throughout this thesis.

I use "Play" as a short term for Play! Framework throughout this thesis.

1.4 List of acronyms

This section lists various acronyms used throughout this thesis (in alphabetical order).

- **AJAX:** Asynchronous JavaScript and XML.
- **API:** Application Programming Interface.
- **CLR:** Common Language Runtime.
- **CSS:** Cascading Style Sheets.
- **DOM:** Document Object Model.
- **HTML:** Hypertext Markup Language.
- **HTTP:** Hypertext Transfer Protocol.
- **IDE:** Integrated Development Environment.
- **IETF:** Internet Engineering Task Force.
- **IIS:** Internet Information Services.
- **IOC:** Inversion of Control.
- **JSON:** JavaScript Object Notation.
- **JVM:** Java Virtual Machine.
- **MVC:** Model View Controller.

---

[2] The data access layer in an application refers to all code that access an external storage unit, often a database.

- **MVP:** Model View Presenter.
- **MVVM:** Model View ViewModel.
- **RPC:** Remote Procedure Call.
- **QoS:** Quality of Service.
- **SOA:** Service Oriented Architecture.
- **WS-polling:** WebSocket-polling.
- **XML:** Extensible Markup Language.

1.5 Code base

The code base for this thesis is located on GitHub [13]. You can also find all raw data for the results from the project's second part there.

1.6 Outline

This thesis is made up by six main parts, most with chapters. The parts mean to make the context of each chapter more clear to the reader. All pages within a part, has the part name as top text.

I give a brief overview of the parts and chapters below. Bold text indicates a part, while underlined text indicate a chapter. Chapters are also accompanied by a chapter number.

Chapter 1: Introduction: The introduction gives an overview of the thesis, its problem statements and terminology as well as related work.

Background: This part contains chapters that describe relevant topics for the thesis.

Chapter 2: The World Wide Web: A short chapter that introduce the reader to some changes the World Wide Web has seen throughout its lifespan.

Chapter 3: Real time: This chapter explains what real time is and how to achieve it with HTTP and WebSockets. It also discusses drawbacks and benefits of both.

Chapter 4: Frameworks for real time web applications: This chapter provides a short introduction to a number of frameworks that exist for real time web applications.

Chapter 5: Server software: This chapter gives a short description of various server software that can be used to host the different frameworks.

Project part 1: Hands on development: This part describes the work done regarding usability of five selected frameworks. It represents the first of the two parts of my project work.

Chapter 6: Methodology: This chapter describes the methodology of the first part of my project work.

Chapter 7: Results: This chapter describes my experiences with each of the selected frameworks.

Project part 2: Load testing: This part describes the work done regarding the load testing of the frameworks.

Chapter 8: Methodology: This chapter describes the methodology of the second part of my project work.

Chapter 9: Results: This chapter presents the results from the load tests.

Chapter 10: Analysis: In this chapter, I discuss the validity of the results in this part. I also provide an interpretation of what they mean.

Conclusion: This part contains summaries of each of the project parts. It also contains the conclusions to each of the problem statements.

Chapter 11: Frameworks: This chapter covers the three first problem statements as well as a summary of all framework-related work.

Chapter 12: WebSockets versus HTTP: This chapter covers the last two problem statements. It also summarizes the work related to comparing WebSockets to HTTP transports.

Chapter 13: Further work: This chapter provides suggestions to projects that can be based on this thesis.

Sources: All sources of this thesis.

Appendix: The appendices of this thesis.

Background

2   The World Wide Web

The World Wide Web has been available for 20 years [14]. But over those 20 years it has changed in almost every thinkable way.

The improvements to the Web have changed the way we use it. Visiting a web page before meant reading a page of text that maybe had some pictures on it. Today, CSS has given web pages a more vivid look with various styling options. AJAX has made them more dynamic. Finally, with HTML5, more revolutionary changes are yet to come.

Along with HTML5 comes a new protocol for the Web: WebSockets. Its purpose is to improve a certain kind of web applications, namely real time applications. A real time application is when clients receive updates from the server as they occur (for more information see chapter 3). Real time web applications have been around for some time, but before they relied on the aging HTTP 1.1 protocol.

2.1   HTTP/1.0

Version 1.0 of HTTP was created in 1996–the World Wide Web's childhood [15]. Besides text, web pages maybe had a few embedded objects at that time[3]. But as the Internet grew, it soon became clear that the user experience had to be improved.

At this time, CSS too was in its childhood [16]. It caught people's attention and more and more browsers started to support it. Embedding a style sheet in a HTML-file adds another object that the client has to download. This is no problem today, but with the HTTP 1.0 protocol it required quite a lot of unnecessary work for both the client and server.

---

[3] Embedded objects consisted mostly of images, but also some early forms of style sheets.

Downloading one element in a HTML-file, or even the HTML-file itself from the server, required one TCP request (Figure 2-1). The server then replied and closed the connection. Throughout the duration of the request, the client waited [17]. Getting a HTML-file with a style sheet and three images then required five consecutive requests in total.
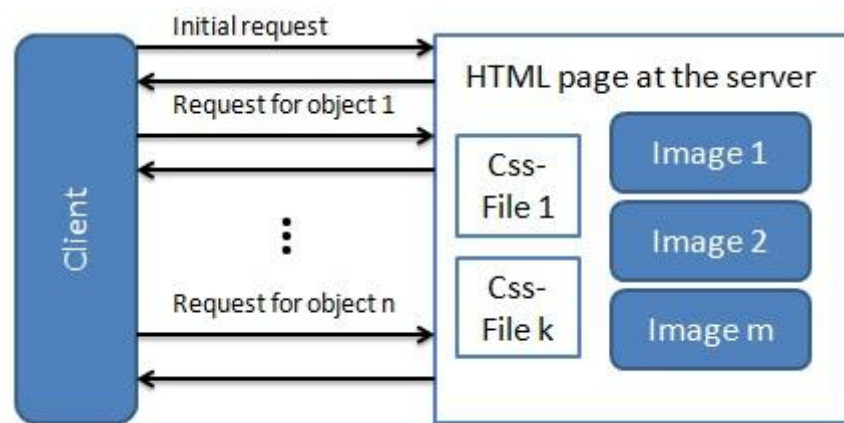
Figure 2-1: Several request to get the whole page.

2.2 HTTP/1.1

After just three years, HTTP/1.1 was released as a standard [18]. It introduced several improvements. One of these was persistent connections. This allowed several request to be made at the same time [19]. It was a dramatic change at the time, as it allowed clients to get several objects concurrently.

Another radical improvement was the ability for a browser to cache parts of an object. This allowed an interrupted download to be resumed later by the help of the cached data. Web applications were also given the possibility of sending chunked data [18], letting servers start sending a response without knowing how long it was. In theory, it could be infinite, as we shall see in section 3.1.3.

The authors of the protocol showed great foresight when they made sure that future protocols could be backwards compatible with HTTP/1.1. The upgrade request-header [17] makes it possible for a client to request the use of another protocol. The server can then chose to upgrade, but it is not required.

Updating from version 1.0 to 1.1 may not seem like a giant leap, but it actually was. Looking at the lengths of the different protocol specifications is an indication of how much more detailed the 1.1 protocol is[4].

---

[4] The HTTP/1.0 protocol is 17902 words while the 1.1 version is 57915 words.

2.3 HTTP/2.0

During the work period of this thesis, a working draft for the 2.0 version of HTTP surfaced. It started out as an initiate from Google called SPDY [20] in 2012. This was then used as the basis for the HTTP/2.0 specification in November that year [21]. The draft has had steady development throughout 2013, and it is planned to be delivered as a proposed standard November 2014 [22].

There are several motivations for a new version of HTTP. Modern web pages contain many elements embedded in its HTML. Today a client parses the HTML and makes requests to get elements as it reads them. HTTP/2.0 allows multiple requests and responses to be sent concurrently on a single connection. The draft referrers to this concept as a stream [21]. With this concept, a server can provide all embedded objects in an HTML page as several responses to a single request (Figure 2-2).
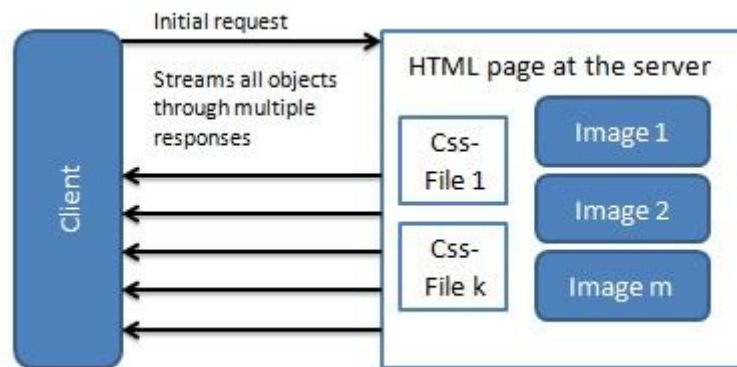
Figure 2-2: Several responses pushed to a client after a single request.

Another vital improvement proposed with HTTP/2.0 is header compression. The goal of this feature is the same as the stream concept. With more concurrency and compression, networks will experience less load. Download times will also be faster, improving the user experience.

Improvements to security are also among the goals of the draft. There are some indications pointing towards a web where unencrypted traffic will no longer exist [23]. It will be exiting to follow the development of the draft.

3 Real time

Real time web applications are not a new phenomenon. Recently, the concept of real time has gotten a lot of attention. There are varying degrees of real time content provided by such an application. At the lower end of the scale, there are for example online comment sections that update whenever someone posts a comment. An example of an application with more real time content is Facebook. It displays notifications and your friends' activities to you as soon as it happens (Figure 3-1).

"As soon as it happens" is exactly what real time is: providing updates for the clients immediately, without the need for refreshing the page on the client side. And as the examples above show, the real time aspect of an application can be either a small feature, or the core concept of the application.

Figure 3-1: Screenshot of real time updates in Facebook.

3.1 The Real time Web with HTTP

Real time works by pushing updates to clients as they appear. This is not how HTTP works–the client always has to initiate the communication. To accommodate the growing need for applications of this sort, several techniques have been utilized. Using HTTP in untraditional ways has been the regular way of accomplishing real time until recently. The introduction of WebSockets may render this deprecated.

3.1.1   Polling

As the first attempt of providing real time updates from a server, polling is a simple approach. It works by having the client make normal HTTP-requests, but at a set interval [24]. The server then instantly sends back a response, either containing new data or just an empty response if there was nothing to retrieve (Figure 3-2). Polling has obvious flaws, like how to set the interval to prevent empty responses while not flooding the server. Therefore, other mechanisms are far more widespread.
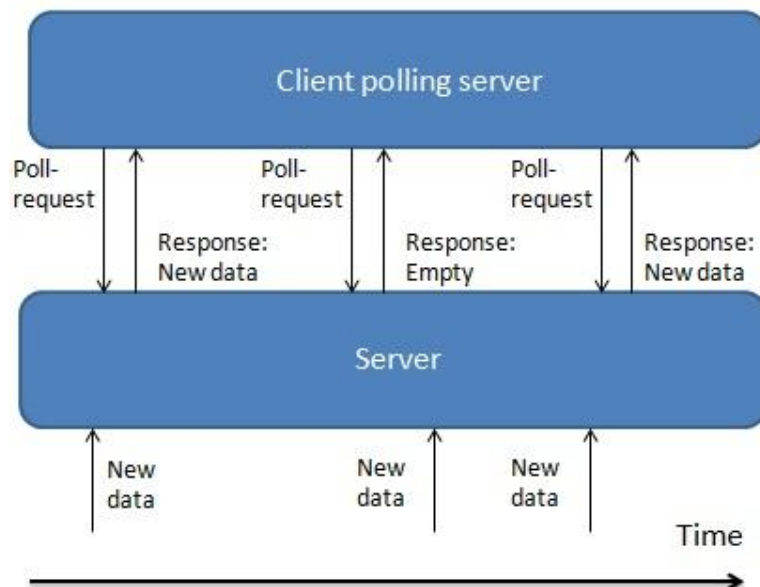
Figure 3-2: Polling.

There is a way to improve a little upon polling: piggybacking [25]. Polling the server at regular intervals is usually done in parallel to other HTTP-requests initiated by client actions. These actions, also get responses back from the server. Piggybacking takes advantage of this by sending updated data back via these responses. In that way, the client may get new data between the polling interval (Figure 3-3).
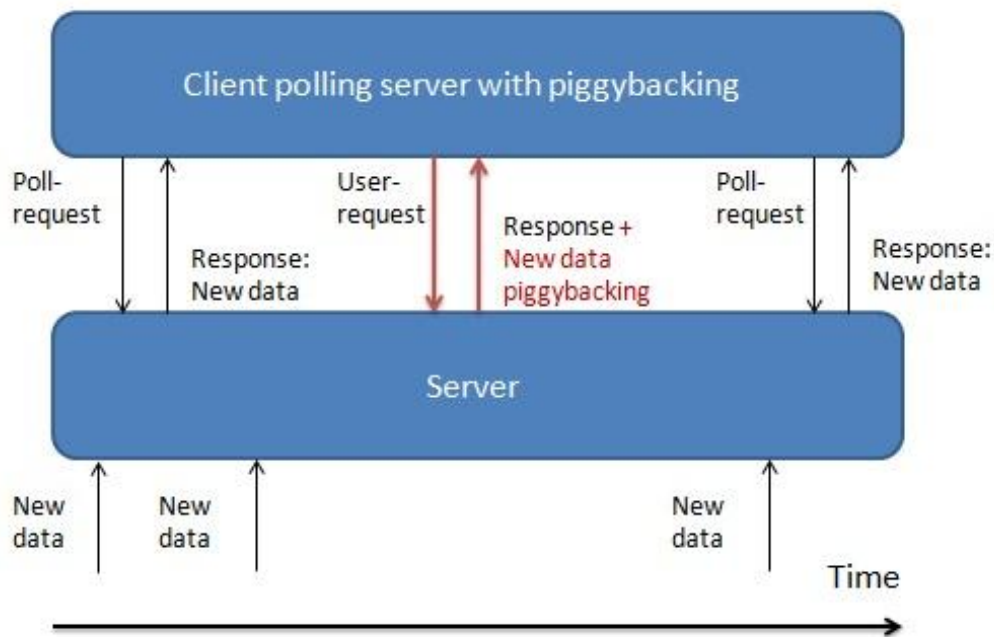
Figure 3-3: Polling with piggybacking.

3.1.2 Long-polling

Long-polling is related to polling. It basically works the same way, but with one rather important difference. By utilizing the keep-alive header in HTTP 1.1, the connection to the server is kept open after the client has made a request [25] (Figure 3-4). This allows the server wait before responding. It cannot do this forever, so eventually it times out. The client then makes a new request [8].
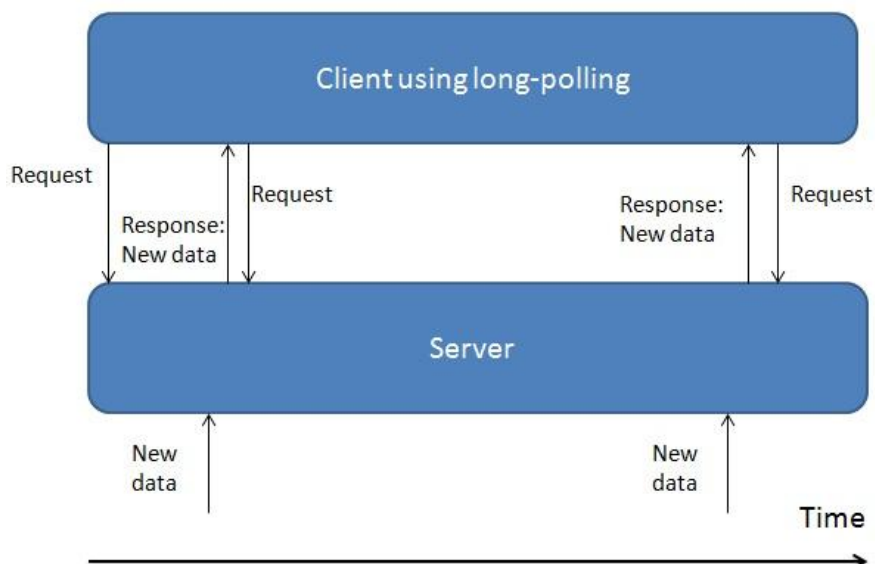
Figure 3-4: Long-polling.

14

3.1.3 HTTP-streaming

HTTP-streaming is an old technique introduced by Netscape as early as 1992 [8]– well before even HTTP 1.0 became standard. Two forms of streaming exist; page streaming and service streaming. Both have the server streaming content in a long-lived TCP-connection. Accomplishing this requires the server to never send the instruction to close the connection. Instead, it remains open throughout the entire course of a client's session.

Service streaming is initialized by a client request. This type of streaming is otherwise known as XHR-streaming [9]. The name comes from the use of a long-lived XMLHttpRequest to send new data. Page streaming uses the initial page request to stream data. This gives more flexibility regarding the lifetime of the connection. Otherwise, the two methods are similar.

A common implementation of HTTP-streaming today is the so-called "forever frame". A forever frame is an iframe that receives script-tags in an everlasting response from a server [26]. Browsers execute code in script-tags when they read it. The server therefore sends data to the clients wrapped up as JavaScript functions within script-tags (Figure 3-5).
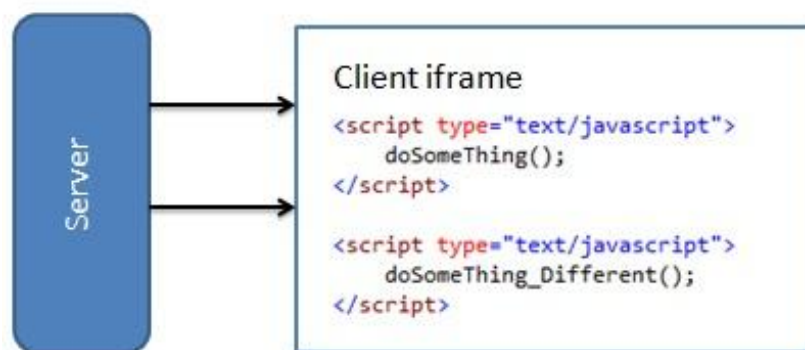
Figure 3-5: A client using the forever frame technique receives script-tags from the server.

3.1.4 Comet

Long-polling and HTTP-streaming are often referred to as Comet or Comet Programming [27]. Comet is an umbrella term that captures different ways to have the server as the initiating part in client/server communication. A rather significant effort has been made to create an official standard for Comet [28], but it has yet to become approved by the IETF. With the introduction of WebSockets, it may never be.

3.1.5   Server-Sent Events

Let's move on into the borders of Web 2.0 with HTML5's Server-Sent Events [29]. Server-Sent Events takes advantage of the "text/event-stream" content type of HTTP/1.1 to push messages to the client [30]. It is, in other words, a one way communication channel from the server to the client.

Still, the client always has to connect first–"subscribe" to the channel. Then the server can send events whenever new data is available. It can keep the connection open forever in theory. However, both the client or any intervening proxies can close it. One can also configure it to close after a given amount of time. The time the client should wait to reconnect is also configurable [30]. Server-Sent Events behavior can thus be quite close to long-polling (Figure 3-6).

Unlike long-polling, developers using Server-Sent Events have access to a simple API [31]. The API gives access to the EventSource interface, which provides straightforward JavaScript code. It allows the server to trigger events in the browser, which updates the content on the client. With the possibility of setting an ID on each message sent, the client can easily reconnect and continue where it left off. The server just need to look up its ID. This makes Server-Sent Events very robust.
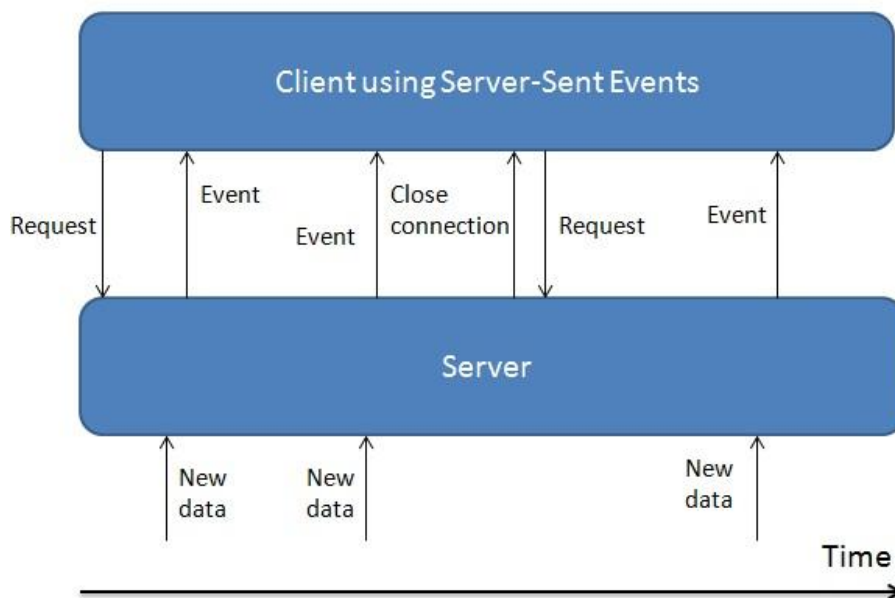
Figure 3-6: Server-Sent Events resembles long-polling.

3.2 WebSockets

We have seen that HTTP/1.1, that came only three years after its predecessor, was a significant step ahead. This protocol had backwards compatibility in mind (see section 2.2 about upgrade request-header), but just now a new version is in the works.
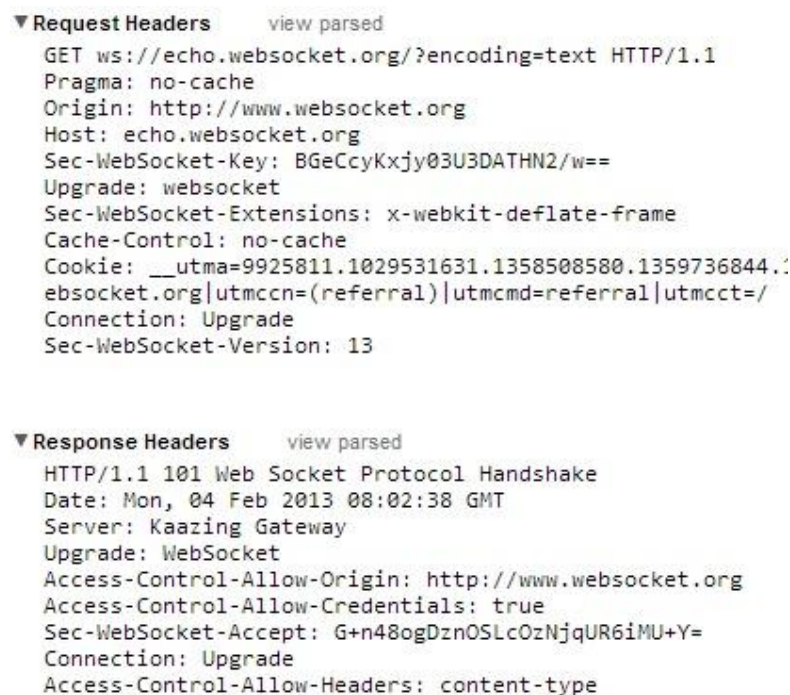
In December 2011, the WebSockets protocol became a proposed IETF specification [32], [33]. The specification document states that the motivation for WebSockets is HTTPs lack of abilities for bi-directional communication between server and client:

> *"The WebSocket Protocol is designed to supersede existing bi-directional communication technologies that use HTTP as a transport layer to benefit from existing infrastructure"* [32] – section 1.1

3.2.1 How it works

WebSockets, like HTTP, uses TCP as underlying protocol. While HTTP needs several "hacks" (see section 3.1), WebSockets provides bidirectional communication right out of the box.

The WebSocket protocol uses the same ports as HTTP and HTTPS (80 and 443, respectively). This allows the initial handshake to be done via traditional HTTP (Figure 3-7). The client states that it wants to use WebSockets, and the server sends a response if it supports it. This ensures backwards compatibility with browsers that don't support WebSockets. Developers can use this to make their applications use HTTP methods instead.

```
▼ Request Headers        view parsed
    GET ws://echo.websocket.org/?encoding=text HTTP/1.1
    Pragma: no-cache
    Origin: http://www.websocket.org
    Host: echo.websocket.org
    Sec-WebSocket-Key: BGeCcyKxjy03U3DATHN2/w==
    Upgrade: websocket
    Sec-WebSocket-Extensions: x-webkit-deflate-frame
    Cache-Control: no-cache
    Cookie: __utma=9925811.1029531631.1358508580.1359736844.
    ebsocket.org|utmccn=(referral)|utmcmd=referral|utmcct=/
    Connection: Upgrade
    Sec-WebSocket-Version: 13
```

```
▼ Response Headers       view parsed
    HTTP/1.1 101 Web Socket Protocol Handshake
    Date: Mon, 04 Feb 2013 08:02:38 GMT
    Server: Kaazing Gateway
    Upgrade: WebSocket
    Access-Control-Allow-Origin: http://www.websocket.org
    Access-Control-Allow-Credentials: true
    Sec-WebSocket-Accept: G+n48ogDznOSLcOzNjqUR6iMU+Y=
    Connection: Upgrade
    Access-Control-Allow-Headers: content-type
```

Figure 3-7: WebSocket handshake as seen in Google Chrome.

Sending messages back and forth once the connection is up, is a lot more efficient than what HTTP can provide. Data in request/response headers in HTTP may accumulate to hundreds of bytes [24], while WebSockets sends messages in frames with only two bytes overhead [34]. Frames can be sent both ways at the same time eliminating the need for more than one connection (Figure 3-8).
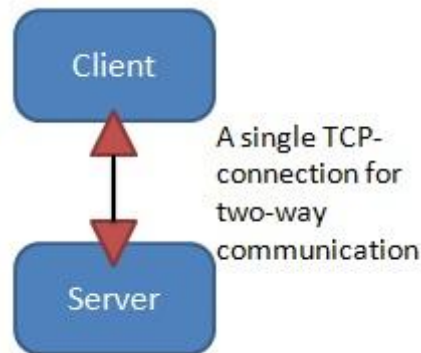
Figure 3-8: Frames can pass both way through one connection.

3.2.2 The WebSockets API

As with Server-Sent Events, WebSockets has its own API [35] that provide the WebSocket interface. This API is a little simpler than the EventSource interface in my opinion. It does not support custom events; just open, close, receiving a message and error.

It provides an easy way to send messages using the send function. Besides, it has an attribute for keeping track of buffered data on the client. This makes the API rather powerful in spite of being simple. Its simplicity is in accordance with the intention of the protocol:

> *"Basically it is intended to be as close to just exposing raw TCP to script as possible given the constraints of the Web."* [32] – section 1.5

3.3 Drawbacks of HTTP techniques

In section 3.1, I gave a rudimentary description of different ways to achieve real time communication with HTTP. They mostly work in the same way, but use some different settings for keeping connections open and pushing messages to the client. Most used is probably long-polling, because it is not affected by intervening proxies. However, there are also some issues that the following sections will describe.

3.3.1 Really real time?

Long-polling builds upon the idea of polling. But whereas polling is a naïve approach, long-polling is smarter. One of the major issues with normal polling is how to determine the polling interval.

Thinking real time, one might want to say that the client should make a new request each time it receives the response of the last. This would create a lot of stress for the server. Dealing with this would require some serious load balancing technology, leading to a expensive solution. A short interval would also increase the number of empty responses if new updates arrive in an inconsistent interval at the server (Figure 3-9).
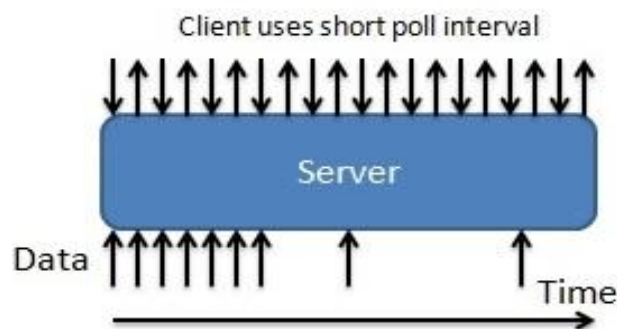
Figure 3-9: Short poll interval gets new data fast, but demands server resources.

With a longer interval (Figure 3-10), the longer it takes before new data arrives at the client, thus making the application less real time. Even with piggybacking, one cannot achieve anything close to real time with a longer interval unless the server receives new data at a regular, known interval. As long as this interval isn't too short, polling may be a good choice. A weather application for instance, might get new updates every hour. Clients can use this knowledge to poll the server accordingly.
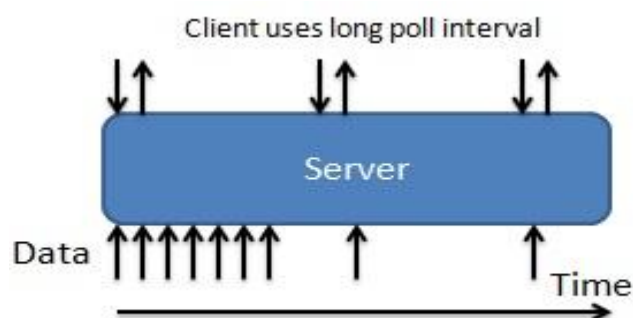
Figure 3-10: Long interval makes the client miss a lot of updates in most cases.

3.3.2   When long-polling becomes polling

As I said, long-polling is a lot smarter than polling. Keeping requests open for a longer time, ensures that the number of unnecessary requests is a lot less than with polling. Though if the server receives updates at a high rate, the connection will never be able to stay open.

Each time the client tries to initiate long-polling, there is always some data waiting for it and the server responds at once [24]. This effect makes long-polling work just as regular polling at a short interval. Comparing Figure 3-9 to Figure 3-11, one can see that long-polling does not outperform polling as long as the server-side updates are frequent.
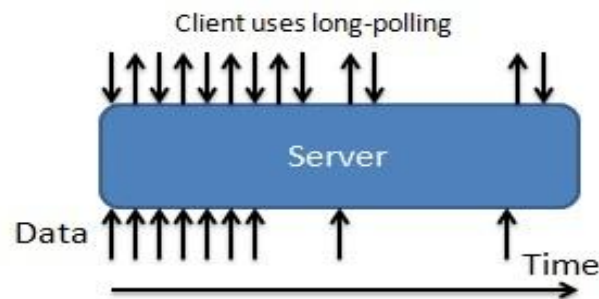
Figure 3-11: Long-polling has no benefit over polling at high update-rates.

Norges Bank Investment Management provided[5] a counter on their homepage, www.nbim.no, that showed the total value of the Norwegian Government Pension Fund. The counter updated several times per second. If each change in that number was a response from the server, it wouldn't matter if it was polling or long-polling in use. The load on their network would be quite substantial in a short time. This little widget, though, faked real time as it polled the server every 30 seconds and got the values from the past 30 seconds.

3.3.3   Streaming techniques

Using streaming techniques is a different approach than having the client poll for data. With HTTP-streaming and Server-Sent Events, the server is the initiating part[6]. You can configure Server-Sent Events to work almost like long-polling (see section 3.1.5), but it uses push just as streaming does.

---

[5] Since this was written, NBIM has gotten new homepages. The counter is no longer there.
[6] After the client has connected, it doesn't need to ask for data. The server streams data as it arrives.

A forever frame allows the server to push updates to the client wrapped up in script-tags. Client-side there has to be some mechanism to make the received scripts do something useful. Receiving new data in an ever-growing DOM-element, also creates some challenges related to memory management. If you don't clear the frame at regular intervals, you will have a memory leak in your application. Using XHR-streaming, you avoid this issue. But you still have to handle a persistent HTTP-connection.

Having an open connection that sends a lot of data, gives rise to another problem: proxy-servers and firewalls [24]. The nature of the HTTP-protocol may cause these to buffer the response, thus creating a lot of latency for the client (Figure 3-12). To avoid this, many Comet-based streaming solutions fall back to long-polling if the server or the clients detect buffering.
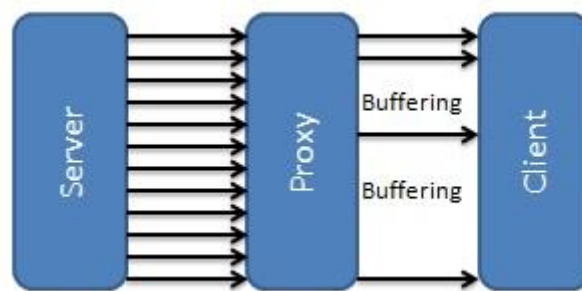
Figure 3-12: Intervening proxy buffering response-stream from server.

A forever frame makes the developer write some extra code to handle the incoming scripts. The EventSource interface gives developers a more powerful toolbox for handling incoming events (see section 3.1.5). Utilizing pure eventhandlers also ensures that there is no need for cleaning up after the incoming data. But, in the end, Server-Sent Events is still HTTP. And as we shall see, the HTTP protocol has issues of its own.

3.4 HTTP was never designed for real time

HTTP/1.1 introduced the keep-alive flag, chunked encoding and persistent connections. Claiming that HTTP wasn't designed for real time in spite of these features, may seem rather presumptuous. The next sections cover what I believe to be HTTP's greatest weaknesses compared to WebSockets for the purpose of real time.

3.4.1 Overhead

In section 3.2.1, I mentioned that headers in HTTP requests/responses can accumulate to hundreds of bytes [24]. Peter Lubbers and Frank Greco made a simple application for comparing polling to WebSockets [36]. To shed some light on the issue with headers, I will borrow some data from their tests. Their simple stock-ticker application polls a server every second to get new data. The counterpart uses WebSockets to get the same information.

In this particular case, the header-data for the polling application accumulates to a total of 871 bytes per request. With just a few clients this is not a lot. But when you have hundreds of thousands of clients, the network throughput increases exponentially. A use case with 100 000 users polling every second means that the server's network has to deal with 665 megabits per second of throughput[7]. Having the same amount of messages in WebSockets creates only a fraction of that. With 2 bytes of excess data in each frame, it accumulates to a mere 1.5 megabits per second[8].

Using polling to represent HTTP against WebSockets is a little unfair in my opinion. Polling is the naïve approach of achieving real time, while WebSockets is designed for it. However, it shows my point: HTTP-headers have much excess data, but most of the time, 99% of this data is irrelevant for a real time application. Achieving a lot less excess data than this example is possible with HTTP through for example long-polling or Server-Sent Events, though nothing will use as little as WebSockets.

3.4.2   *Unidirectional*

HTTP was finished in the 90s and it is still going strong. It is rather impressive, but it is not unnatural that something that old will struggle with new trends. WebSockets is a protocol designed only for the purpose of bi-directional communication [32]–HTTP is not. In fact, no matter how you look at it, or how you try to hack, HTTP remains unidirectional.

As a result, real time applications with HTTP have to use several TCP-connections (Figure 3-13). Even with Server-Sent Events, one will need one connection to push the events to the client and at least one for the client to send messages to the server. With HTTP 1.0 some browsers used several TCP-connections to get more concurrent loading of web pages [19]. Now the same work-around is used to achieve simulated bi-directional communication. And as with last time this was the case, we need an improvement, namely WebSockets.
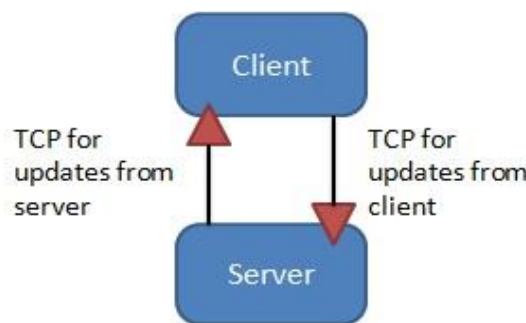
Figure 3-13: Two TCP connections simulating bi-directional communication.

---

[7] 87 100 000 bytes * 8 = 696 800 000 bits / 1024^2 = 665 Mbits
[8] 200 000 bytes * 8 = 1 600 000 bits / 1024^2 = 1.526 Mbits

3.5 WebSockets is still young

With new technology comes the almost everlasting issue of backwards compatibility. The use of the HTTP upgrade request-header ensures this for WebSockets. Implementing it, though, would have been a lot easier if all browsers supported it. As I write this, Internet Explorer has about 14% [37] of the browser market with IE8 and IE9 as the most dominant [37]. None of these support WebSockets. The rest of the major browsers does support WebSockets. But it will be several years before developers can safely assume that every single user out there uses a browser with WebSockets support.

To deal with this, applications have to fall back to other, supported techniques. In turn, this leads to more code. Luckily, frameworks like SignalR [38] and Socket.IO [39] (see chapter 4) abstract this away for developers. However, sometimes you want more control over the software you create than a framework supplies. And even with frameworks, you might end up having to do some workarounds for certain clients, where the fallbacks provided by the framework don't suffice.

3.5.1 Know when to use it

Writing an application with some real time elements is a different task than writing a full-blown, dynamic, real time application. Examples of the two is an online newspaper with a live comment section and a chat room, respectively.

In the first case, using WebSockets may provide more pain than gain. Updates in a comment section do not need to be instant. Having the client poll the server will not change the user's perception of the application. Chatting is a different matter. Especially in a chat room, where several people talk to each other at once. This makes real time crucial to the user's perception of the application. In turn, this makes it worth the extra effort of providing fallbacks for the browsers that don't support WebSockets.

3.5.2 Know how to use it

An important thing to realize is that WebSockets is not HTTP/2.0. It is a standalone protocol designed to fill the gap of HTTP regarding bidirectional communication. Failing to understand this might cause developers to use WebSockets in applications that don't need it. An informative webpage, like Wikipedia, will probably not benefit from using WebSockets. You get less overhead in request-headers, but your application will have to serve mostly idle connections (Figure 3-14). The only real server to client communication is when the client request a new page.
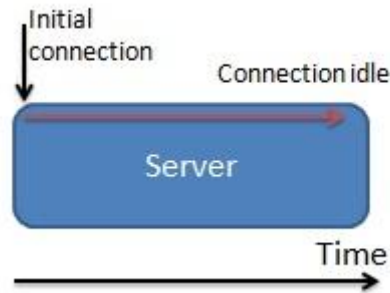
Figure 3-14: No need for exchanging messages leaves idle connections.

Understanding your application's environment is another vital aspect. WebSockets should handle proxies and firewalls gracefully [24]. But you might still encounter some problems. Especially if the traffic between your server and the client has to go through an older proxy along the way. Peter Lubbers indicates this in a blog-post from May 2010 [40], and even though this post is rather old, it might be a problem for some. His suggested way of handling the issue is the use of a secure connection (wss:// instead of ws://). In my opinion, this is a good practice, since it makes data encrypted.

3.6 The use of real time

The World Wide Web has seen many innovations throughout its lifespan. Each time something new comes around, it is hard to determine if it has come to stay. It is always a question of need. Real time is no different from any other new developments: it has to be useful. And without some form of establishment throughout the web, nobody will ever notice it.

There is no doubt that real time content is useful in many aspects, and that in others it is even crucial. An auction site with time based auctions is dependent on delivering the latest bids to all users. Forcing their clients to refresh a web page manually to see the latest bids, would render it completely useless. On the other side of the scale we find web sites that utilize real time to provide their users with a greater sense of convenience. Getting your friends' status updates immediately can hardly be seen as crucial, but it does enhance the user's perception of the experience.

Another interesting development is the increasing use of real time content provided by web sites that are more static. Most of this has to do with integrating social content like live comment-sections, trending articles and such. Again this is purely to make the content seem more dynamic and make the experience better for the users.

Looking at pure web page usage of real time, it is, with some exceptions, about the user's experience. But if we expand our perspective a little, it soon becomes clear how much of an impact real time might have on our lives in the future. Live video streaming is a common feature. But the technology is still in its youth, with buffering issues and broadband capacities as bottlenecks (Figure 3-15). As the technological aspects evolve, I believe we will see a lot more usage of live video streaming across the web. Presumably, WebSockets, with its ability to stream binary data [24], will play a central part in future improvements to video streams.
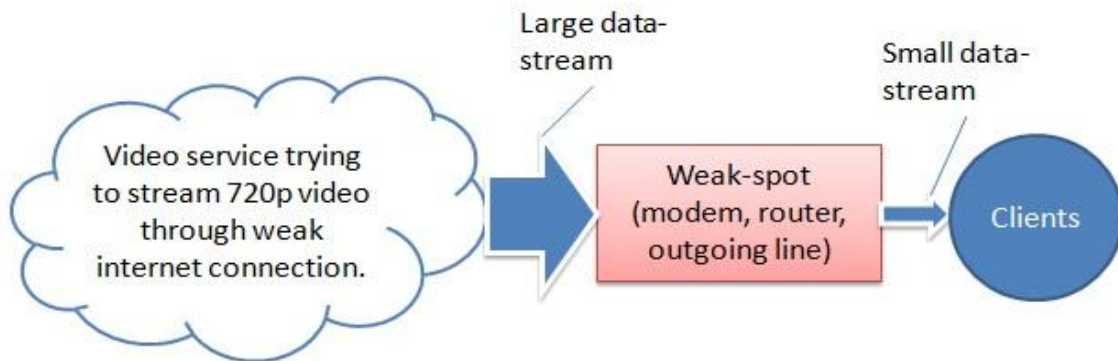
Figure 3-15: Hopefully, this scenario will be a thing of the past in a not too distant future.

3.7 Conclusion

We have seen that WebSockets is superior to HTTP when it comes to bi-directional communication. Sometimes, it is not necessary with a bi-directional channel to achieve real time content. In some cases, HTTP-streaming techniques may be better than WebSockets. This is the case if most of the communication is from server to client, and the amount of header-data in the HTTP protocol is no cause for problems. If proxies may cause issues you can use long-polling instead.

Looking at these aspects leads me to say that HTTP methods may still be a better choice than WebSockets for some real time purposes. But ignoring backwards compatibility, there is no getting away from the fact that WebSockets is superior to HTTP for real time applications. After all, that was why WebSockets was created in the first place. Still, HTTP, with Server-Sent Events in particular, remains a strong alternative if you only need real time push. I believe that the other techniques will be outdated in a couple of years. WebSockets will replace them with some use of Server-Sent Events applications as well.

Background

I believe that in the future, when WebSockets has been around for a long while, most applications will prefer it. Furthermore, my opinion is that any future versions of HTTP will not incorporate WebSockets. The two will remain what they are: two separate things.

Social networks like Facebook, collaboration tools like Google Docs and other real time use cases are already widespread, and that will most likely not change any time soon. Real time is here to stay, which is good since it provides vast, and yet unseen, possibilities.

4 Frameworks for real time apps

Many frameworks for real time web applications exist, and it would be close to impossible to introduce them all here. I have chosen to present the ten that, in my opinion, are the most widespread and interesting. Six of these are featured in either part 1 or part 2 of the project work[9].

4.1 SignalR

SignalR is a library for the ASP.NET framework that strives to make it easy for developers to add real time functionality to applications [38]. It has clients for both web applications, Windows Phone, ordinary Windows Store apps, other Windows applications and even Android an iPhone apps[10].

The library provides two separate levels of abstraction for real time: Hubs and Persistent Connections. Persistent Connections are the most low level, extending the WebSocket API with a few events for reconnection and a few other things [41]. Hubs introduces even more abstraction on top of Persistent Connections. These utilize a RPC[11] API that allows for simple real time programming.

4.2 Socket.IO

Socket.IO is a library for Node (see section 5.3 for information on Node) that *"aims to make real time apps possible in every browser and mobile"* [39]. Since it runs on Node, it only uses JavaScript as a language. Available clients are browsers, either mobile or desktop. It also has a client library for Node (meaning that both the client and the server can use Node).

Using an event driven architecture, it closely resembles the WebSocket API. Socket.IO extends upon this by letting developers name their own events as well as using the traditional WebSockets events (see section 3.2.2 about WebSockets events).

4.3 Atmosphere

The Atmosphere framework is built for Java servers [42]. It uses a JavaScript client that supports all major browsers [43]. It provides an API that is more low level than the Hubs API of SignalR, but a little higher than the Persistent Connections of the same library.

On the client, Atmosphere provides something that looks very much like the WebSocket API, only with a few other events.

---

[9] One of the five frameworks in part 1 was switched out in part 2.
[10] In order to write C# code for Android and iPhone, one has to use the cross platform framework Xamarin [142].
[11] RPC is when a client calls a method situated on a server directly. For more information see rfc5531 [143].

4.4 Sails.js

Sails.js is a MVC framework for Node [44]. It has full real time capabilities through the use of Socket.IO. The way this is accomplished, is rather interesting. Through what the developers call *"transport agnostic routing"* [45], controllers are allowed to automatically handle Socket.IO messages. Traditionally, this is something that is handled by separate code as with for instance SignalR in an ASP.NET MVC application.

4.5 Play! Framework

The Play! framework, or just Play, is a MVC framework written in Scala and Java [46]. It is built on Akka, a framework that uses the Actor Model[12] to help developers make distributed applications [47].

Play is real time enabled, meaning that it has constructs to help developers make real time functionality. These constructs are a lot closer to bare metal than what is provided by pure real time libraries and frameworks.

4.6 SockJS

SockJS is a JavaScript library for web browsers that gives developers an object that resembles the WebSocket interface. This object can be used to connect to a server. Server-side, there is a Node implementation as well servers for Erlang and Python [48]. Other servers are under development.

4.7 Meteor

Many different types of web application frameworks exists, but none of these are similar to Meteor. It promises to simplify the process of making web applications drastically [49].

Real time is the heart of Meteor, and it uses SockJS to enable this. All code with Meteor is written in JavaScript, but interesting enough, it is not a Node framework (even though it runs within a Node container behind the scenes [50]).

4.8 Lightstreamer

Weswit is the Italian company [51] behind the oldest real time framework I've been able to find (released in 2000 [52]). Lightstreamer is the name of the framework, and from the looks of it, it is also the most comprehensive in what it offers of different APIs.

---

[12] An Actor is a construct used to form hierarchies of entities that handle concurrency through messaging [144].

The Lightstreamer server itself is written in Java, but through the use of an adapter based model, you can make server-side code with JavaScript, C# or Java. There is also a long list of client-side APIs including a JavaScript client for web browsers [53].

4.9 Planet Framework

Planet is a real time framework for Python [54]. It is built with an emphasis on the development of social real time websites. The framework also provides its own IDE[13], specially designed APIs and scaling mechanisms.

4.10 XSockets.NET

XSockets.NET is an event driven library for .NET [55] that adopts a model that somewhat resembles that of Socket.IO on the client. Server-side it uses an approach that looks like the controllers of ASP.NET MVC.

The library supports WebSockets and has fallbacks in order to support all major browsers.

---

[13] An IDE is best explained as an advanced text editor designed for writing code. It usually offers various support mechanisms like code completion, refactoring, debugging and keyword highlighting.

5 Server software

All web applications have to be hosted on a server with the help of some sort of software. Various server software exist for different languages. In this chapter, I give a short description of some that are relevant for the thesis.

5.1 ASP.NET

Most applications for ASP.NET use IIS as an application server. The current version of IIS is IIS8 which was released alongside Windows 8 and Windows Server 2012 [56]. With it came support for the WebSockets protocol [57]. IIS is free as it is a part of the Windows operating system[14]. It is a Windows feature that has to be activated.

For development, it isn't always necessary to deploy an application to IIS. Along with the ASP.NET framework and Visual Studio, developers get access to IIS Express. As with all products in Microsoft's Express series[15], it is a lightweight version of the original. It provides what you normally need while developing, but its performance is not enough for production.

5.2 Java

Java applications can run on a number of different servers on any operating system. The most common Java servers are Jetty, JBoss and Tomcat [58].

Jetty is an open source server hosted by the Eclipse foundation [59]. It is considered to be very lightweight, but it has support for a wide variety of features.

JBoss is a product suite that includes several server types. The JBoss application server and Netty are probably the most known. Play uses the latter as its application server[16].

Tomcat is an open source server from Apache [60] made for Java applications. Another server from Apache, the Apache Web Server, is used by the Planet Framework as preferred deployment server [61].

---

[14] The operating system itself is not free.
[15] Visual Studio Express is one of these products.
[16] This can be observed when the Play server throws an Exception. It is often of a type within the org.jboss package.

5.3 JavaScript

Node is a new and exciting piece of technology that, for the first time, allows for server side JavaScript. It is based on Chrome's V8 JavaScript runtime, and one of its primary goals is to help build fast, scalable network applications [62]. A web server is a fast, scalable network application. In other words, Node itself isn't a web server, but it provides simple mechanisms for writing one [63].

5.4 Writing your own in C# or Java

There is nothing wrong with building your own web server as you do with Node. In its simplest form it is just a main method with some mechanism for listening to incoming requests. The following example is from the "Getting Started" tutorial for Microsoft's OWIN [64].

```csharp
class Program
{
    static void Main(string[] args)
    {
        using (Microsoft.Owin.Hosting.WebApp.Start<Startup1>("http://localhost:9000"))
        {
            Console.WriteLine("Press [enter] to quit...");
            Console.ReadLine();
        }
    }
}
```

Example 5-1: Using OWIN to make a server.

This isn't completely bare metal, but it is fairly close. Within the "using" clause there is some sort of loop that listens to requests. Another class can handle these incoming requests, thus giving you a web server.

Project part 1:
Hands on development

6   Methodology

This part will focus on the first three problem statements (see section 1.1). I selected and compared five different frameworks for real time web application development, based on usability. This part covers the results of this comparison. The focus is from a programmer's perspective. A simple web application was therefore implemented with each framework. Section 6.2 describes the application.

The selection of frameworks made in this part, is the foundation for the second part. There, each will be compared based on performance (read more in the second part of the project from page 71).

6.1 Selection criteria

As described in chapter 4, several frameworks for real time applications exist. Many have similar programming interfaces and features, but for the purpose of this thesis, only a few was selected. Due to the timeframe of this thesis, I chose five frameworks. Each selected framework must support the criteria listed in this section. A detailed justification of the selected frameworks is given in section 6.7.

6.1.1   WebSockets support

The framework must support WebSockets. Each framework went through load testing (in part 2), where the individual transport mechanisms were compared as well as framework performance. In order to answer if WebSockets is an improvement to HTTP, each framework had to support it.

6.1.2   Fallback support

For the same reasons as why WebSockets had to be supported, at least one HTTP-transport must offered as well.

6.1.3   Presentation

The home page or GitHub repository of the framework should look presentable and give at least basic documentation and/or tutorials. Any piece of technology is hard to use if you cannot find out how to use it. Lack of proper documentation makes a considerable negative impression.

6.1.4   Maturity

Real time applications has been around for many years [65], but most of the frameworks for real time web applications are a lot younger. Maturity should still be given consideration, but if a frameworks offers something unique and potentially revolutionary, it may still be selected even if it is immature.

6.2 Description of test application

An auction house was implemented with each of the selected frameworks. The application has the following requirements specification:

- Users must receive real time updates regarding all global events.
- Global events are defined as all actions except from logging in and registering a new user.
- Users must be able to register an account and log in.
- Users must be able to add and remove items.
- Users can only remove an item added by themselves.
- An item does at least have the following properties: name, minimum price, info about who added it and who has the leading bid.
- Users must be able to place bids on all items, including their own.
- If the framework does not specify a specific template language or other means of creating a user interface, the application will utilize a common user interface implemented with Knockout [66] [17].
- MySql will be utilized as database unless implementing it requires substantial workarounds, that may cause the framework to misbehave.
- All applications should have tests covering the most critical aspects of the program logic.

As the purpose of the application is to demonstrate real time features only, there are certain elements that a normal web application like this would have had. This includes session management, security (passwords are plain text in my app) and elements related to the auction, like timers and bid history.

6.3 Discussion of use cases

Registering and logging in are actions that follow a traditional request/response pattern. These two use cases are present in order to test a framework's capabilities regarding such communication. The remainder of functionality uses broadcasting; sending the response to all connected clients. This is the most crucial functionality of a real time application.

Another form of real time communication is so-called peer-to-peer communication (client to client) via the server. Implementing such functionality is basically the same as basic request/response, only that the response is sent to another client. As this adds little extra complexity, I regard it as unnecessary to test this aspect in this thesis.

---

[17] Knockout is a MVVM framework for JavaScript and HTML [66].

6.4 Evaluation

Evaluating each framework was done during and after the development of the test application described earlier. The evaluation was from a programmer's perspective, shedding light on how the framework is to work with rather than how it performs. In order to get a complete picture, this process followed a preset list of criteria described below. For consistency, the criteria are used as subheadings for each framework in the results chapter (chapter 7). The term "user" in the following subsections refers to a programmer.

6.4.1 Documentation

It is probably the most visited page in a programmer's browser history while he/she is working with a new piece of technology. How the documentation is written and structured can make a lot of difference when it comes to a user's experience. The code can be simple enough, but that means nothing if many frustrating hours are wasted looking for reference in the documentation.

Tutorials and examples are one of the most effective means to help a user get started with an unfamiliar technology. The presence of such was therefore considered very positive. Other than that, the documentation was evaluated based on structure, simplicity and content.

6.4.2 Simplicity

A real time web application framework should act as a communication layer in a web application. Other functionality such as session management, database operations and authentication, are normally already present server-side in a web application. I evaluated whether a framework offers too much functionality or not.

Serialization and deserialization of data is central in server/client communication, regardless of real time or not. With JavaScript as the client language, the most common data exchange format is JSON. In my opinion, a framework should handle this behind the scenes, so that the user can focus on implementing the communication part of the application.

Keeping track of connected clients can easily lead to errors, and if you do it in an inefficient way, it can also cause performance issues. Each framework was evaluated based on how it handles this problem. The more abstraction the better.

Abstracting this away from the user is usually a good thing, but it depends on how the clients are offered back to the user. If it requires complicated code just to send a message, the client handling can be as foolproof as it wants–it does not make the overall experience better. I therefore also looked at what constructs each framework offers for sending messages to clients.

6.4.3 Maintainability

Being able to write maintainable code has become alpha omega in modern system development. Any framework that introduces unnecessary complexity and dependencies between entities, makes maintainability difficult. In some cases, there may be possible to write maintainable code even if it is very dependent upon other entities. Such scenarios may render unit testing impossible, but with proper tools, one can test entities using integration testing instead. Normally, you would prefer to have both unit- and integration tests as well as other forms of testing (for instance functional). I measured maintainability based on how the natural structure of the application is with each framework and how that impacts testability.

6.4.4 Browser support

This criteria is directly linked with what transports a framework offers. WebSockets is not supported by older browsers, neither are some of the fallbacks. How a framework detects what transport can be used is crucial for the overall experience. A proper real time framework should handle transport selection gracefully in the background.

It should be possible to choose a "lower lever" transport[18]. For most use cases, this is not a demand, but considering the work I did in the second part of the project, it was important to me.

6.4.5 Maturity

A lot of real time frameworks are a work in progress and have not yet reached a stable version. Such frameworks often change drastically, causing users to change their applications completely to keep up. Hence, these frameworks are not suited for production code.

Another way of measuring maturity can be to look at projects that use a framework. If nobody is using it, there is often a reason. Furthermore, one can look at the amount of bugs and errors that appear during development. Errors directly related to the framework's core, are often a sign of immature code. This is also often a sign of unmaintained code, which many would say is even worse than immaturity.

6.4.6 Other criteria

There may be other criteria that some deem more relevant than the ones listed above. All those I have listed are based on what is most important in my opinion. Other criteria that I considered were:

- Coding environment: What IDEs a framework allows you to work with can make a difference for some. Especially regarding support for debugging and code analysis. I believe that these things matters less if a framework is a suitable tool for a job.

---

[18] WebSockets is the highest level possible, whereas polling is the lowest.

- Community: A large community can be an indication of a mature framework. It does not guarantee it. Furthermore, a small community does not mean that the framework isn't good.
- Programming language: When wanting to implement a feature into an existing application, one often want use the same programming language. Also when building something from scratch, it may matter. However, measuring usability of frameworks based on what programming language they use, will be heavily influenced by my subjective opinion. While all frameworks are evaluated based on my opinions, I wish to keep my preferences regarding programming language out of the evaluation.

6.5 Limitations

There are some limitations regarding what kinds of frameworks that are suitable for this thesis.

6.5.1 Cloud based solutions

If my work only consisted of comparing the usability of different frameworks, cloud based solutions could have been a part of it. But due to the load testing aspect, I cannot test cloud based frameworks. It is impossible to have an equal test between a framework, whose server I do not control, and a normal framework running on a local server.

6.5.2 Rapid development

Many frameworks for real time web applications have appeared over the last couple of years. Most of these are not among the best, and almost all are very early in the development process. Because of this, there may be some frameworks not considered for this thesis that suddenly have become one of the "buzz-words". An example of this is the Java-framework Atmosphere (see section 4.3).

6.6 Other choices

This section describes other choices related to this part of the project.

6.6.1 Common user interface

The application that was implemented was the same for all frameworks. It looks the same, and the functionality is the same. To keep it as simple as possible, I tried to share as much of the user interface code as possible. For that purpose, I have created a common user interface using Knockout that I used when possible. Some frameworks came bundled with some other way of creating the user interface. These frameworks did not make use of the common user interface.

6.6.2 Choice of database engine

What database the application use, is not relevant for the real time aspects of it. Nonetheless, the application I built has a database component. To keep this as similar as possible, MySql was used for each framework that supported it. All these application used the same database.

6.6.3 Functional testing

As each application appears the same to a browser, I made a common functional test case. This worked for all the frameworks that used the same database. The tests use Selenium WebDriver [67] to drive real browsers through a series of tests. Java was used as implementation language for these tests.

6.7 Selected frameworks

I selected the five different frameworks for some different reasons. There are three "pure" real time frameworks and two "real time enabled" frameworks. A pure real time framework has real time as its only functionality. A real time enabled framework has other main functions, but can offer real time functionality. This section describes why each of the five were selected.

6.7.1 Socket.IO

Node is rapidly gaining popularity. Using JavaScript on the server is an exciting thought, and the notion is changing the way developers think of the language. Because of this, it was only natural to choose at least one Node based framework.

Several libraries and frameworks exist for real time with Node (see chapter 4). Of these, Socket.IO stands out from the crowd. It seems to have the largest community, and it gets a lot of attention.

Socket.IO has its own homepage that has some documentation. It is not a lot, but everything is presented in an orderly fashion. The project uses GitHub for further documentation and wikis. The library is very lightweight, which explains why there isn't a lot of documentation.

The library offers some fallbacks to ensure compatibility with all browsers. WebSockets is the preferred transport. If it is not supported, Socket.IO will fall back to one of the following transports [68]:

- Adobe Flash Socket, which uses Flash to establish a TCP connection between server and client.
- Ajax long-polling
- Ajax multipart streaming (Http-streaming)
- Forever frame
- JSONP Polling. Polling with JSONP allows for cross-domain requests.

Currently, the library is in version 0.9, but a 1.0 release is around the corner. Nonetheless, it is considered stable and is used by several projects[19]

---

[19] Examples of use: Trello, an online cooperation and project planning tool [145]. Sails.js, a web application framework with real time in its core (see section 4.4)

6.7.2 *Lightstreamer*

Lightstreamer is completely different from all the other frameworks I have found. It is a commercial product from a rather large, European company with customers worldwide.

Being operational since 2000, it is the oldest framework in this thesis by far. The more modern frameworks have different approaches to real time than Lightstreamer. Seeing how these new ideas play out against many years of experience, will be interesting.

Weswit is the name of the company behind Lightstreamer, which is their only product. This means that all the customers they list, are using Lightstreamer. With names like NASA and Sky on the list [69], it is obviously a trusted and presumably a mature product.

The framework seems thoroughly documented. Separated documents for each API as well as some general concepts, leaves a good impression.

Lightstreamer supports the following fallbacks when WebSockets is not available:

- Http-streaming.
- Http-polling.
- Polling with WebSockets can also be selected.

6.7.3 *Play! Framework*

One of the questions I seek to answer is whether you need a framework at all to implement real time applications. Play offers some help for real time, but not a lot. It is close to working with bare metal real time development.

Not many web application frameworks I have seen promotes real time features like Play does. Hence, it stands out from the crowd in this matter. The framework has become more popular recently, and some serious actors are using it in production[20].

As a developer, you get two helper classes for real time functionality. One for WebSockets [70] and one for Comet (Http-streaming, [71]).

A clear answer to the initial question will be if the development process with these helpers, turns out to be cumbersome. Then, bare metal will most likely be even harder.

---

[20] See Play's homepage [46] at the bottom.

6.7.4   SignalR

SignalR is one of the few libraries made specifically for ASP.NET. Additionally, it is amongst the libraries that have gotten the most attention in recent time.

Two developers on the ASP.NET team started the work on SignalR in 2011[21]. The fact that Microsoft supports the project is thus no surprise.

SignalR builds on concepts that are familiar to .NET developers, like the use of IOC containers [72] and Nuget [73].

The library uses an interesting form of abstraction in its programming model. The use of Hubs and RPC will be very interesting to get a closer look at.

WebSockets is the preferred transport. If it is not supported, SignalR will fall back gracefully to one of these transports:

- Server-Sent Events
- Forever frame
- Long-polling

I also have to mention another reason for choosing SignalR. My first experience with real time technologies was through a project with SignalR in 2012. It really sparked my interest. Without this experience, I would probably have written my master thesis about some other topic.

6.7.5   Meteor

Meteor is not a completed framework. Nor is it close to completion. Nonetheless, I have chosen to test it due to a number of reasons.

First of all, it is radically different from any other similar framework. It uses JavaScript both on the client and server, which is possible through the use of Node. It is not a Node framework, though.

Meteor tries to share most of its code between the server and the client, blurring the line between them. I intend to find out if this is an application model that is better than the traditional.

Another interesting feature of Meteor is how real time is closely integrated with its core. Actually, a lot of features seems tightly coupled to the core as of now. MongoDB is the only supported database, even though support for others is planned [74]. I implemented the test application using MongoDB. There is an unofficial add-on that allows you to use MySql [75], but I did not utilize this.

---

[21] First commit to GitHub was in 2011 [146].

Project part 1: Hands on development

Windows is currently not supported by Meteor. An unofficial fork[22] of the project exists for Windows [76]. I used this for my test. The only real difference between this and the official, is the command line tool that has to be compatible with Windows.

Meteor uses SockJS [50] under the hood to perform its real time updates. SockJS had support for WebSockets, but Meteor first received support for it a couple of days before I started my work with the framework [77].

If WebSockets is not supported, SockJS falls back gracefully to one of the following transports [78]:

- Server-Sent Events (for the browser Opera only)
- Http-streaming (iframe or Xhr depending on the browser)
- Long-polling
- Polling (just for really old browsers)

---

[22] Forking a project means that you use that project as a base for your own. See https://help.github.com/articles/fork-a-repo for more information.

7 Results

This chapter gives an in-depth description of my experiences with each of the frameworks. Each section starts with an introduction describing some specific choices made for the given framework. For instance what server was used or whether it used some uncommon approaches. The subsections of each describe details from the development process based on the criteria described in section 6.4.

7.1 Socket.IO

Socket.IO is a library for Node that use an event based approach to real time. As most examples with Socket.IO show it in conjunction with Express, I used this in my test application. Express is a web application framework [79]. The only aspects of the application that use Express is the server itself, including serving static files. The Socket.IO application makes use of the common user interface described in section 6.6.1.

7.1.1 Documentation

With Node already installed, downloading and installing Socket.IO into a project is simple. One simple command is all you need: "npm install socket.io". While it goes without saying that you need to have Node installed, I think there should have been a link in the documentation to where you can get it nonetheless.

Documentation is not something Socket.IO has a lot of. Considering the size of the library, this is not surprising. What it has, covers everything in enough detail. Frequent use of examples, makes it easy to read and understand.

The structure isn't perfect. Some pieces reside in the readme on GitHub [80], while you find other pieces in the wiki [81]. There are some logic behind this, with the API documentation in the readme, and other aspects in the Wiki, but there is no obvious flow when browsing it.

As of writing, the current version of Socket.IO is 0.9. The documentation states that it is for the "upcoming" 1.0 release. For the entire duration of the work with this thesis, this has been the case, but the version has yet to be released. I have run into no problems regarding this, which leads me to believe that most of the API is set already.

All the examples are very small, which is good for readability. I missed some larger examples, though. Something with more than one HTML-file and a little more complex functionality, would have been beneficial.

7.1.2 Simplicity

In the spirit of Node[23], Socket.IO is lightweight. Classical web application elements include authorization and session management. Both of these can be tricky to handle with real time frameworks, but Socket.IO provides some simple mechanisms [81]. It is actually oblivious to sessions, leaving it up to the server library you use to handle this [82].

Some frameworks and libraries offer a lot of configuration. Usually this is a good thing, but sometimes it is easy to get lost in translation. Socket.IO has many options that you can tweak, but it is far from needed [81]. You manage settings using code instead of one or more files. I prefer the latter, but since the language is JavaScript, one can simply store configuration data in a JSON file and parse it at run-time.

A common use case for a web page is to have several, separated real time features. IGN.com[24] has functionality to show current readers of articles, as well as real time comment sections. Socket.IO allows developers to register different channels, or "rooms". This makes implementation of such functionality simple.

Sending data back and forth is a dream with Socket.IO. Behind the scenes, it uses JSON. As a result, you can send a normal object and receive it in the callback on the other end (Example 7-1).

```
On the server:
socket.on('registerUser', function(user){
    service.registerUser(user.username, user.firstname,

On the client:
socket.on('placeBidResponse', function(bid){
    itemView.placeBid(bid.itemno, bid.value,
```

Example 7-1: Serialization of objects happens behind the scenes.

You have to be cautious about sending Date-objects, though. A common problem with JSON, is that Date objects don't deserialize too well. Instead of deserializing as Date-objects, they appear as strings [83].

---

[23] According to Node's homepage [62], its event-driven model makes it lightweight and efficient.

[24] See www.ign.com/?setccpref=US (American IGN). IGN.com is an entertainment website with focus on video games, movies, music and TV series.

Project part 1: Hands on development

Another nice feature is that you don't need to relate to the concept of a client. Instead, you deal with either one or multiple sockets (Example 7-2).

```
Send to caller:

socket.emit('registerItemResponse', itemno);

Send to all:

io.sockets.emit('deleteItemResponse', itemno);
```

Example 7-2: Simple response and message broadcast with Socket.IO.

Because of this simple abstraction, sending a message never requires more than a single line of code. Socket.IO also provides constructs for sending messages to a specific client. To do this you need the id of the particular client's socket. Associating this with for instance a username, has to be handled manually.

A final note on features offered by Socket.IO is that it closely resembles the WebSocket API. While this API only has four events, Socket.IO lets you define your own. These act as a further separation of the "onmessage" event. The result is a code structure where the flow of the application shines through without the need to dig too deep.

7.1.3   Maintainability

Socket.IO follows the programming principles of Node and provides an event based model. While all code written for Node can be testable, testing events is a little tricky. The event driven architecture conceals all logic regarding sending and receiving messages within callbacks. Luckily, there are ways to work around this.

One option is to separate all code within the callback to its own module (Example 7-3). Then you can write tests for this module in separation, as it does not know that it is an event that calls it.

```
socket.on('placeBid', function(data){
    service.placeBid(data.itemno, data.userId, data.va
        if(!error) {
            io.sockets.emit('placeBidResponse', bid);
        } else {
            console.error(error);
        }
    });
});
```

Example 7-3: Separate logic within callbacks to separate module.

Another option is to put the callbacks themselves in a separate module. This makes the code a little less readable in my opinion, which is why I went with the first solution.

To test that specific callbacks execute as expected, one has to use either integration or functional testing. As Socket.IO provides a client library for Node, this is simple to achieve without too much complications (Example 7-4). This method can also be used for unit testing purposes by injecting mocks and stubs into the module with the Socket.IO logic. In its essence, it will still be an integration test since you have to start the server, but at least you get to test modules in isolation.

```
before(function(done) { //set up socket.io client
    socket = client.connect('localhost', {port: 80});
    socket2 = client.connect('localhost', {port: 80});

    socket.on('connect', function () {
        console.log("Socket connected");
        done();
    });

});

Testing broadcast:

it('should broadcast bid', function(done) {

    socket.on('placeBidResponse', function(bid) {
        bid.userId.should.equal(1);
        ...............................................
    socket2.on('placeBidResponse', function(bid) {
        bid.userId.should.equal(1);
```

Example 7-4: Testing events with Socket.IO.

Modules can be challenging to keep small. Large modules, just as large classes in object oriented languages, are harder to maintain. This also applies to the routing of events with Socket.IO. Even if you keep the code within each event's callback short, it can quickly become a mess if you have many events. Using the namespace construct can help provide a stronger separation of concerns in such cases. However, this also introduces extra overhead, which leads to the fact that there is no perfect solution to this problem.

7.1.4 Browser support

As promised, the library supports all major browsers with no quirks. Transport mechanism is selected automatically during the handshake process of a connection. However, it states that it supports HTTP-streaming, but I did not get this to work. After some investigation, I found out that it has been removed from the core [84].

7.1.5   Maturity

GitHub claims that Socket.IO version 1.0 is the "upcoming" release, which has been the case for over a year. Commits to the repository has been varying and it had a long dead period. This dead period seems to coincide with another project from the same people: Engine.IO (see commit pages for Engine.IO and Socket.IO on GitHub [85], [86]).

Engine.IO is a more low level implementation of Socket.IO. Socket.IO is actually an extra abstraction layer on top of Engine.IO. It is reasonable that the creators would wish to separate the most low level functionality into its own library. This makes it easier for other developers to build upon it to make other frameworks.

That there are no more commits to a project is a classic sign of a "dead" project. With the activity on Engine.IO and recent activity on the Socket.IO project, I do not think this is the case for Socket.IO. Nonetheless, little activity does not mean that the product is immature. In this case it is completely opposite, as Socket.IO is stable and well suited for production environments. Since the documentation for the 1.0 release is already out, one can trust that no major changes will come soon.

There is the question of Node itself, though. It too hasn't reached version 1.0, and the community surrounding it isn't the largest. As a result, the community surrounding Socket.IO is even smaller. Many questions on Stack Overflow[25] is about Socket.IO however, a clear indication that it is widespread. The tendency with Node is that the community is growing, and more and more developers see it as a technology for the future–an opinion I share.

7.2 Lightstreamer

Lightstreamer is a commercial product from an Italian company named Weswit. For the work in this part I used the free version they offer [87].

In a real life scenario, the Lightstreamer server would only handle real time aspects. My application uses it as a web server as well. It also has database communication. As this is not the intended use of the Lightstreamer server, I have not written about this in the following sections. The Lightstreamer application does not make use of the common user interface described in section 6.6.1.

7.2.1   Documentation

Getting started with Lightstreamer is a simple and well documented process [53]. The rest of Lightstreamer is also extremely well documented. It is, by far, the most comprehensive documentation of all the frameworks in this thesis. With such a large scale, it is clear that Lightstreamer is a framework rather than a library like Socket.IO.

---

[25] Stack Overflow is a question and answer website for programmers. See www.stackoverflow.com.

With a lot of documentation comes a great responsibility to organize it. Weswit does this well. Each of the many APIs has its own document, while a common document covers all general concepts. These concepts should maybe have been given more room in the documentation. The documentation only offers a single page to one of the most central aspects of the framework: the different subscription modes[26]. I didn't get a good sense of what the difference between these was before I found a forum post that explained it [88].

Another shortcoming is the almost complete lack of tutorials–only one exist [89]. That would have been okay if the samples were well documented and well written, but this isn't so. Many samples accompany the framework (Figure 7-1), and these illustrate different uses. Understanding them at a conceptual level is easy enough, but when you start digging into the code, trouble starts.
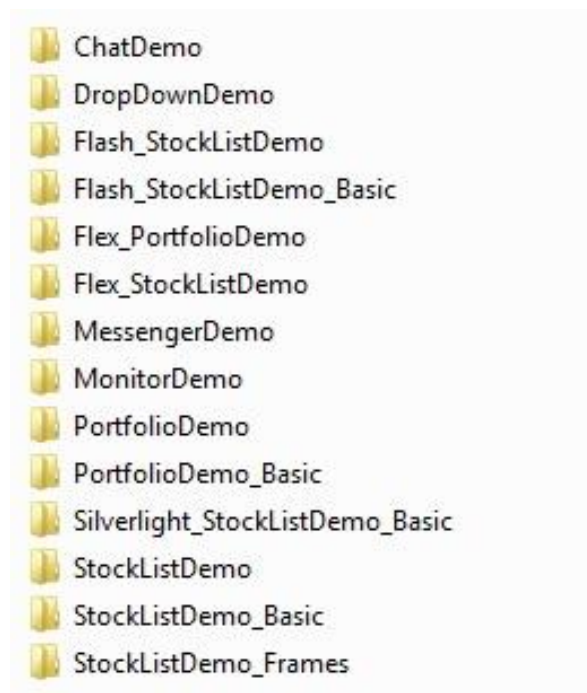
Figure 7-1: The contents of Lightstreamer's "demos" folder (for JavaScript client)

The samples are "documented" through comments in the code, and these are not abundant. Furthermore, the code is rather messy and hard to follow. I had a hard time figuring out what parts was related to Lightstreamer and what regarded the user interface. As a result, I spent many hours debugging both the client- and server-side code in order to understand what was going on.

---

[26] The subscription modes are RAW, MERGE, COMMAND and DISTINCT.

Lightstreamer's server comes with a large configuration file where you can tweak its performance to fit your needs. There isn't too many options, but the file is easy to get lost in. This is because Weswit chose to write all documentation regarding the various options within the file itself as comments. As a result, it is a lot larger than what it could have been. Some XML elements are also hard to spot because they are commented out.

7.2.2 Simplicity

Compared to the other frameworks in this thesis, Lightstreamer is huge. It even feels larger than Play and Meteor (see sections 7.3 and 7.5), which both are full featured web application frameworks. Socket.IO provides real time features alongside a web application, running on the same server. This is not the intended use of Lightstreamer. It is meant to be a stand-alone server rather than a layer in an application stack.

Other parts of the application (other servers) interact with Lightstreamer through an adapter infrastructure. There can be any number of data adapters and one so-called metadata adapter. Data adapters handle subscriptions and sending updates. The metadata adapter handles incoming messages from the clients, session management, authorization and QoS [90]. This is a nice separation of concerns, but it also introduces some complexity. There has to be some form of connection between the adapters, meaning that they either have to depend on each other, or have some common dependency. (Figure 7-2).
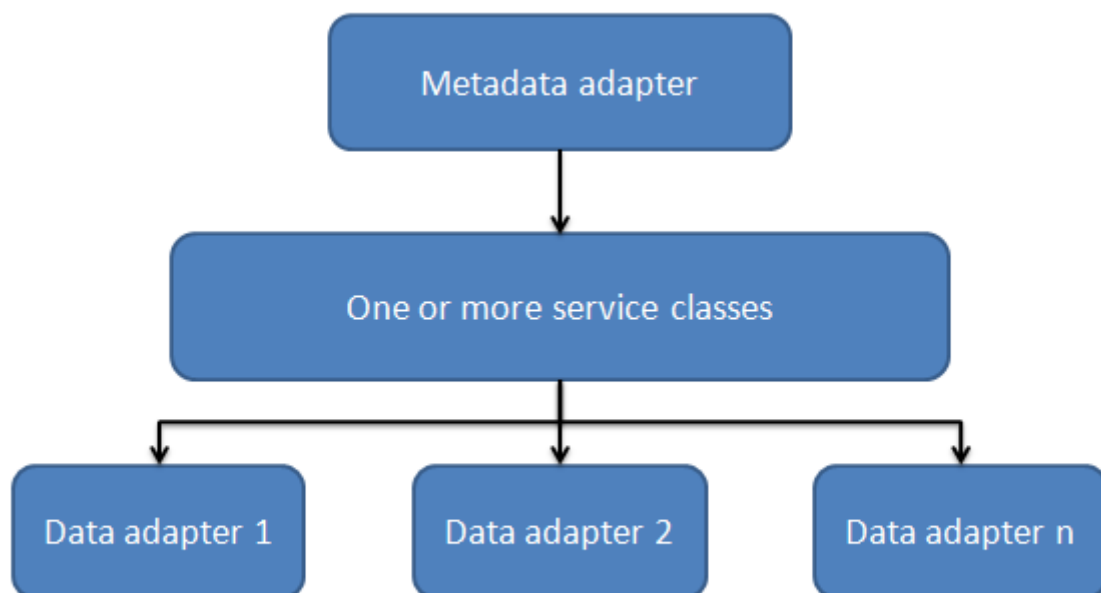
Figure 7-2: How the various adapters can interact through a shared service layer.

Project part 1: Hands on development

Lightstreamer uses an application model that resembles a SOA [91]. More specifically, it uses a publish/subscribe model. It doesn't follow all SOA principles, though [92]. Some examples of this is that it is not discoverable and it has a strong coupling between the clients and the server.

Clients subscribe to different items rather than listening to certain events. To me, this feels a little old fashioned, as it creates a very tight coupling between the DOM and the items. The items, which "live" on the server, need to have fields that corresponds to the fields in the DOM on the client (Example 7-5).

```html
<span data-source="lightstreamer" data-field="key"></span> <br/>
    <span data-source="lightstreamer" data-field="price"></span>
```

Example 7-5: Tight coupling to the DOM.

One can work around this by using a "message" DOM element with only one field. If this field receives updates in JSON format, you can mimic an event driven architecture. There are several drawbacks to this technique. Serialization and event routing have to be handled manually. The first is somewhat manual no matter (Example 7-6), but converting to JSON adds another layer of complexity. It also makes you unable to benefit from the different subscription modes.

```java
private void placeBid(Object handle, Bid bid) {
        HashMap<String, String> update = new HashMap<String, String>();

        update.put("key", String.valueOf(bid.getItemno()));
        update.put("command", "UPDATE");
        update.put("bid", String.valueOf(bid.getValue()));
        update.put("highestbidder", bid.getUsername());

        listener.smartUpdate(handle, update, false);
    }
}
```

Example 7-6: Serialization of data bound for clients.

Lightstreamer is not intended to be used in this way, and doing it may influence performance. You will be better off using the subscription modes. These are actually powerful, allowing for updates of single fields, deleting and adding items. However, it is a lot more complex than the more "modern" approach given by for instance Socket.IO and SignalR (see sections 7.1 and 7.4).

The tight coupling to the DOM makes using popular MV*[27] frameworks like Angular [93] or Knockout [66] little useful. As a result, it may be hard to integrate Lightstreamer into an existing application where such frameworks are present. With a new application, Lightstreamer can fall through because of this in my opinion. Either that, or the application has to have a clear separation between the Lightstreamer-parts and the rest.

As I mentioned earlier in this section, there has to be some connection between the metadata adapter and the data adapters. Directing an incoming message to its destination is something you have to handle yourself. The same goes for concurrency. The following example shows the flow in my simple test application from an incoming message is received, to a broadcast is sent (Example 7-7).

```java
@Override
public void notifyUserMessage(String user, String sessionID, Str
                          ↓
private void handleMessage(String message) {
        String[] parts = message.split("\\|");
        if(parts[0].equals("ADD")) {
                itemSubscription.addItem(parts[1]);
                          ↓
public synchronized void addItem(String json) {

                          ↓
@Override
public void onAdd(Item item) {

                          ↓
private void add(PrettyItem item, Object handle, boolean snapshot) {
        HashMap<String, String> update = new HashMap<String, String>();
                          ↓
                       Clients
```

Example 7-7: The flow of a message from it is received to a broadcast is sent.

Lightstreamer is broadcast by default. Given its old age, it is not unnatural that the main use case is push. In fact, it performs best if it can function purely as a publisher of events. It would be even better if some other entity than the clients functions as producers of events. This would let the Lightstreamer server do only push, which is what it seems to be most suited for.

---

[27] MVVM, MVC and MVP are referred to as MV* [147].

A clear indication of this is the fact that the default update mechanism is broadcast. To send to individual clients, even back to the caller, you have to have a separate subscription for each client. Request/response features are, in other words, not Lightstreamer's strong suit.

7.2.3 Maintainability

Despite the complexity, there is nothing stopping you from writing maintainable code. Both data- and metadata adapters are based on interfaces, so you can mock them in tests. If an application manipulates the data injected to the adapters by the Lightstreamer server, this is also testable. Since the server uses method injection for this, it is just a matter of creating some test data and inject it in the test.

The samples use an ExecutorService [94] to handle concurrency. Testing methods that use this can be a little messy, but no more than testing callbacks of Socket.IO's events. As long as you can inject a mock of the event listener that the ExecutorService triggers, you are good to go. In other words, the seemingly preferred method for Lightstreamer, is the same I used for Socket.IO (see section 7.1.3).

Client-side on the other hand, is an whole other story. When testing this code, the first thing you'll want to do is to mock out dependencies to Lightstreamer. But before that, you need to understand how this code works. With only a minified version of the code available, this is easier said than done. What Lightstreamer does client side is somewhat advanced, so mocking its behavior isn't the easiest of tasks.

Still, as long as you keep a firm line regarding separation of concerns, on both server and client, you should be able to write maintainable code. After all, Lightstreamer's intended use is to be a separate server in your application stack. Keeping this in mind, you would want to keep this part as lightweight as possible. Other aspects, like database handling and such, should be left out of the Lightstreamer server. But even with all of these precautions in place, the real time specific code of Lightstreamer will be more comprehensive than most other real time frameworks.

7.2.4 Browser support

Lightstreamer supports all major browsers without any trouble. Fallbacks are handled gracefully behind the scenes so that the optimal transport is always used. One thing I have trouble understanding, is the transport mechanism that uses polling with WebSockets. As far as I can tell, there is no real benefit to this as any browser that supports it, also supports streaming with WebSockets.

7.2.5 Maturity

With 14 years of experience, it is safe to say that Lightstreamer is a mature product. In a way, it has become too mature. 14 years ago the focus of real time applications was pure push. This is still what Lightstreamer does best, as the mechanisms for receiving input from clients are somewhat complicated.

Lightstreamer uses a completely different approach for real time than any of the competition. The push orientation is probably due to its old age. But the use of a separate server for all real time functionality is not a bad thing. Depending on the needs of an application, this technique may be the better no matter what framework or library you are using. With Node based solutions for example, the most likely approach is this, since very few applications use only Node as server technology.

Weswit is actually one of the pioneers for real time technologies [95]. Their main problem now is to keep up with modern day trends. The competition comes from lightweight libraries rather than large scale frameworks. In my opinion, they would benefit a lot from making a "light" version of Lightstreamer that introduces a higher level of abstraction. Using for instance an event driven model, this would result in a simple library. Decoupling it from the Lightstreamer server may also be beneficial. This would make it more attractive for Java developers that want some simple real time features in their application.

Nonetheless, the company has a large customer base, and they have received a lot of appraisal [96]. Changing something that many people use and love, isn't always a smart thing. The expression "If it ain't broke, don't fix it" exists for a reason. What I think they should do, is keep everything they got, but also offer what I suggested above.

7.3 Play! Framework

Play is a MVC framework with real time capabilities. I used version 2.1.1, which was the newest at the time. Play comes with a server-side template language for rendering HTML. I minimized the use of this in order to use the common user interface described in section 6.6.1.

7.3.1   Documentation

Obtaining Play is a simple and well documented process [97]. Following installation, the documentation provides a step-by-step guide to get you familiarized with Play. There are also two tutorials: one simple and one more comprehensive [98]. These are mostly easy to follow, but I ran into some issues with links to certain files that didn't exist anymore.

As Play lets you write code in either Java or Scala [98], [99], it is a good thing that the documentation is separated accordingly. The exception is the getting started part, but this covers no code. Splitting this as well would have been a duplication rather than a separation. With the separation, developers can focus on the language they want and have no knowledge of the other.

Play is an open source framework with a rather rapid development process. In addition to referring you to a change log, Play's documentation lets you browse previous versions. This simplifies the process of migrating to a new version.

The framework builds on many other technologies from third parties. Play's documentation does a good job of pointing the reader to resources regarding these technologies. It also explains them, but not in too much detail.

Using examples, the documentation is easy to read and learn from. Learning by doing is, in my opinion, the best way. With Play, there was little need for diving into the source code of the samples it offers. I just followed the examples.

7.3.2   Simplicity

Play is a MVC framework. Hence it resembles most other frameworks in this genre. Still, it has some interesting features that separates it from others:

- A route configuration file that gives you IntelliSense[28] in the editor. It gets compiled at run-time, providing you with feedback just as with code files. You have to manually list all routes. Compared to the route configuration of ASP.NET MVC, where routes follow a pattern, this is a little elaborate.
- Hot code push allows you to run the server once. Each time you change a file, the code recompiles and your page refreshes.
- The template language for HTML is Scala. It resembles the template language of other frameworks like Razor for ASP.NET. Compilation errors from all template files, are also shown as with other files (Figure 7-3)
- Play lets you use Akka Actors to handle concurrency. Actors is an old, mathematical concept [100], but it has become increasingly popular recently. As with the rest of Play, you are not forced to use it.

Figure 7-3: Compilation error with Play.

---

[28] IntelliSense is a support feature in IDEs that provide automatic code completion.

The real time features of Play are close to bare metal, but it provides some nice abstractions. There are two utility classes, one for WebSockets and one for Comet (HTTP-Streaming with forever frame). The "WebSocket" class gives you two channels: in and out. As WebSockets is bidirectional, this is reasonable (Example 7-8).

```
public static WebSocket<JsonNode> wsAuction() {
    final long cid = ctx().id();
    return new WebSocket<JsonNode>() {
                @Override
                public void onReady(WebSocket.In<JsonNode> in, WebSocket.Out<JsonNod
                    try {
```

Example 7-8: Plays WebSocket provide an "in" and an "out" channel when it's ready.

With the in-channel you get access to the "onclose" and "onmessage" events of the WebSocket API. "Onopen" is handled by the "main" WebSocket class (the onReady event in Example 7-8). The out-channel handles sending of messages only.

Since HTTP-streaming is from server to client only, the Comet class has no in-channel. A separate route has to be set up to handle incoming messages. Using a POST route is the most applicable. Outgoing messages use a similar out-channel as the WebSocket class provide. One handy thing the Comet class provides is a callback for disconnects (Example 7-9). With this, along with the out-channel and incoming POST requests, you have a similar API for Comet as for WebSockets.

```
comet.onDisconnected(new Callback0() {
        public void invoke() {
                System.out.println("Comet browser disconnected!");
                getContext().self().tell(messages.newDisconnect(cid),
        }
});
```

Example 7-9: The Comet class provides an event for disconnection handling.

On the client you stand without support. To access WebSockets, the standard WebSockets API is used. Comet depends on the presence of an iframe on the client in order to receive messages. The utility class on the server wraps outgoing data into a function within a HTML script-tag. You specify the name of the function, but it has to be globally available or provided an absolute path[29].

---

[29] An example of an absolute path to a function is: window.someGlobalObject.someGlobalFunction().

Both of the utility classes allow for sending either strings or Jackson JsonNodes[30]. JSON is the natural choice since it allows for easy access on the client. Play even has a helper to extract a request body as a JsonNode, which fits well into the real time stack of the framework [98]. There are some drawbacks, as serializing more complex objects yields more overhead (Example 7-10) Many nested arrays and objects, result in many loop constructs. Overall, this can lower the performance of the application.

```
ArrayNode itemsNode = om.createArrayNode();
for(PrettyItem item : items) {
        ObjectNode itemNode = Json.newObject();
        itemNode.put("name", item.getName());
        itemNode.put("description", item.getDescription());
        itemNode.put("highestBidder", item.getHighestBidder());
        itemNode.put("minPrice", item.getPrice());
        itemNode.put("bid", item.getBid());
        itemNode.put("addedByID", item.getAddedByID());
        itemNode.put("itemno", item.getItemno());
        itemNode.put("expires", item.getExpires().getTime());
        itemsNode.add(itemNode);
}
event.put("items", itemsNode);
```

Example 7-10: A lot of work to serialize complex objects to JSON with Jackson.

Besides the manual fallback handling and serialization, you also have to keep track of clients yourself. A hashmap is probably the best solution for this, as it gives good performance for looking up single clients.

No matter what you use, you need a reliable source of an unique ID for each client. Luckily, Play gives you a connection ID that can be used for this purpose. This ID becomes the key in the hashmap. The value in the hashmap has to be some kind of entity that contains the mechanism for sending messages. I solved this by separating all common methods into an abstract super class. Then I made wrappers for the WebSocket out-channel and the Comet class that inherited from this super class (Figure 7-4). With these two transports only, this is a simple solution. If I had introduced long-polling as well, it would probably have been a little more complicated.

---

[30] Jackson is a library for JSON handling in Java [148].
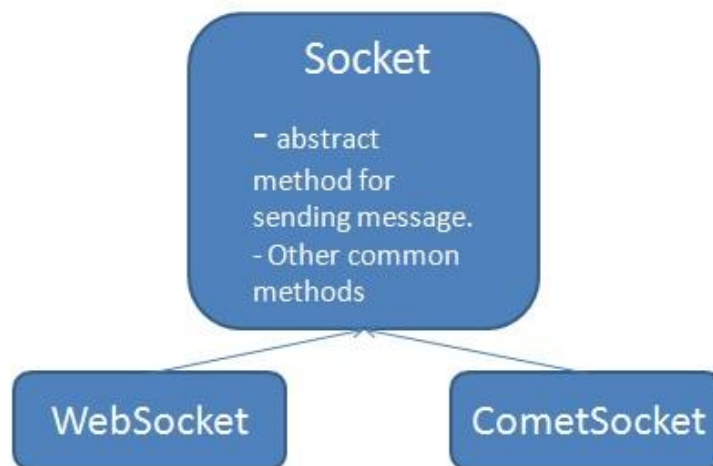
Figure 7-4: Socket helper class hierarchy.

Using a framework with a bare metal client implementation gave some interesting insight into the cost of extra dependencies in a project. I looked at the client-files containing only code regarding communication. In addition, I looked at how many bytes was added to the file "viewModels.js" compared to the size it had in the "TestApplications /AuctionHouseUI" folder on GitHub [13]. This showed that Play's client side communication code adds up to 4898 bytes.

SignalR's client code is 29598 bytes, but since this depends on JQuery, you get another 92639 bytes, adding up to 122237 bytes. Socket.IO's client code has no other dependencies and adds up to 32384 bytes.

I did not do this comparison with Lightstreamer or Meteor. Lightstreamer's client library alone takes up 246107 bytes, so the other files don't really matter—it is huge compared to the others. Meteor is a web application framework with a programming model that makes a comparison like this unfeasible. Communication code often also handles database operations with this framework (see section 7.5). This makes it hard to compare the impact code specific to communication has to file sizes.

7.3.3   *Maintainability*
Writing clean and maintainable code with Play is mostly the same as for any other good framework. There are some small issues, though.

The utility classes for WebSockets and Comet are both based on events. They use Java's counterpart of JavaScript's anonymous callbacks (Example 7-11). This leads to the same issue as the anonymous callbacks of Socket.IO's events. With Socket.IO I proposed a number of solutions (see section 7.1.3). These can also be used for Play. Another aspect where this problem surfaces, is when working with Actors. You can build huge hierarchies of Actors that work together. Testing this is not straight forward, but again, the above solution is a simple workaround.

```
                    Anonymous callback, JavaScript
$.each(clientItems, function(i, item){
    itemView.addItem(item);
});
```

```
              Anonymous callback, Java
in.onClose(new Callback0() {
    public void invoke() {
        System.out.println("Websocket brov
        instance.tell(messages.newDisconne
    }
});
```

Example 7-11: Anonymous callbacks in JavaScript and Java.

All applications should have tests that work from end to end. An event based architecture makes this even more important. Play offers some utilities to help with this issue. You can make a fake server and use Selenium to do functional tests against it

Play also supplies a simple way to use an in-memory database to test your models. A traditional Java web application using for example Spring MVC would have a separate data access layer. Play encourages the use of Ebean [101], which puts data access into the models themselves. Testing is similar, as a normal process with Spring MVC is to set up an in-memory database just as with Play. This process in Spring require a lot of configuration, while Play gives you a single method to call [102]. Play also offers methods to help you test controllers, templates and even the routes, meaning that the hole stack is testable.

7.3.4   Browser support
Using just WebSockets and Comet provided support for all major browsers. Handling fallbacks manually can be achieved in two different ways. I chose to use the user agent string to determine what browser the client was (Example 7-12). In retrospect, this was not the best solution. Instead I could have the client find out if it has one of the WebSocket or MozWebSocket (for Firefox) objects. If not, it doesn't support WebSockets. This information could then be used to connect to the server. I used this approach in the load tests (as you can see from the source code on GitHub [13]).

```
public static boolean hasWebSockets(String useragent) {
        return useragent.contains("Chrome") || useragent.contains("Firefox")
        || useragent.contains("MSIE 10") || useragent.contains("Opera");
}
```

Example 7-12: Code to determine WebSockets support based on browser type.

7.3.5 Maturity

The framework was introduced in 2009, but Play as it is today came out in 2012 as version 2.0 [103]. With it, the core had a major overhauling. Scala support came through an external module in the 1.x versions. The 2.0 version integrated Scala with the core, providing full support for the language.

One might say that the framework matured a lot with this change in the core. It made it easier to provide full support for both Java and Scala as well as providing more consistency in the way you build an application. But, this also means that the core is quite new and unproven. Furthermore, Scala is a relatively new language and it isn't used that much. The TIOBE Programming Community Index, places Scala as number 48 in April 2014 [104].

Nonetheless, Play keeps gaining popularity, and it can be considered a stable piece of software. This is my opinion, but there were some issues. Outdated versions of third party software and broken links on the homepage were the most prominent.

There is no arguing that Play follows modern trends. With support for CoffeeScript, LESS and other popular languages and tools, it offers a lot of freedom. It may not support the same stability as for instance ASP.NET or Spring, but it strives to make development clean and simple.

7.4 SignalR

SignalR is a library for real time with ASP.NET. A common environment would feature SignalR in a web application hosted on IIS. But this part of the thesis does not require high performance of the server the application runs on. Hence, I chose to use IIS Express 7.5. An ASP.NET MVC 4 application wraps the SignalR components. The application uses the common user interface described in section 6.6.1.

SignalR offers two levels of abstraction: Persistent Connections and Hubs. As Hubs are most high level, and what most people are likely to use, I focused on this aspect only in the test application.

7.4.1 Documentation

SignalR's documentation has a very comprehensive introduction part. It talks you through all from what it is to making your first application. As with most other .NET applications, you use Visual Studio as IDE and Nuget as package manager. The documentation does a good job describing how to use these with SignalR.

During early development, the only documentation was on GitHub [105]. Even so, there was still a lot of documentation and many, simple examples. There are still some topics that are covered only on GitHub, but most of the documentation now resides in one place [38].

The new documentation has a lot of tutorials and examples. It is way more comprehensive than what it was on GitHub. Just as Play's documentation does, SignalR's documentation lets you browse previous versions, though not with the same granularity[31]. The content itself is also separated into logical bulks with lots of examples. A nice feature is that all class names appear as links to the class reference[32].

SignalR is the only framework that covers IOC in its documentation. This is probably because it is the only framework that has its own dependency resolver. It is therefore only natural to tell you how to use this or swap it for something else. Still, the term "IOC" isn't even mentioned in most of the other frameworks' documentations.

There are some things I missed in the documentation. Hubs are thoroughly documented, but Persistent Connections are not covered in detail. The GitHub pages covers this, so it may be in the making for the new pages. Furthermore, I missed some documentation regarding testing hubs. Finding information about this required me to search elsewhere [106].

7.4.2 *Simplicity*
The developers made SignalR to be compatible with ASP.NET web applications. Whether you use ASP.NET MVC, WebForms or Self Hosting, using SignalR is the same experience. It is compatible with mechanisms for authentication, IOC and sessions provided by ASP.NET. A developer only has to find out how to match those things with SignalR. This is an easy task as there are many examples for almost everything.

SignalR comes in two different forms: Hubs and Persistent Connections. Persistent Connections are a lower level of abstraction than Hubs. It resembles the WebSockets API, but with some extra methods for handling sending messages, broadcasting messages, reconnecting, groups, etc. A benefit of using this API is that you can access the real time part of an application from several places. This allows for instance a controller action method to broadcast data to clients.

The Hub API strives to be very simple, providing a RPC model which makes the code simple and understandable. A Hub exposes all its public methods to the client and they can be "called" directly (Example 7-13).

---

[31] There is one documentation for the 1.x versions and another for 2.x. Play has more separation, with documentation for both 2.0, 2.1.1 and 2.2.0 for example.
[32] Class references in C# are similar to Javadocs for Java.

```
On the client:
var getUsersBids = function(userID) {
    return auctionHub.server.getUsersBids(parseInt(userID));
};

On the server:
public IEnumerable<ViewBid> GetUsersBids(long userID)
{
    return _service.GetUsersBids(userID);
}
```

Example 7-13: SignalR's RPC model makes it simple and understandable.

C#'s convention is to have method names with a capital cased first letter, while JavaScript has small cased first letter. As a result, the name of the method or function you "call", doesn't match the actual name[33]. It takes a little while to get used to, but in the end it makes sense to follow conventions.

Hubs manage clients through an abstraction that is the most straightforward and easy to understand I've seen. You have two main choices on how to get data from the server to the clients. One is by returning from a method. This sends the returned data back to the caller, just like a response to a request. On the client, the returned data is sent to the "done" callback of a promise, just like a jQuery AJAX-call. Clients can also specify functions that are callable from the server in the same RPC style as clients call server methods (Example 7-14). Furthermore, the "Clients" object also has constructs to access the Caller of a method and to access groups. Groups are similar to Socket.IO's namespaces (see section 7.1.3).

```
On the client:
$.extend(hub.auctionHub.client, {
    receiveItem: function (prettyItem) {
        var expires = new Date(prettyItem.expires);

On the server:
var prettyItem = _service.AddItem(item, username);
if(prettyItem != null)
    Clients.All.receiveItem(prettyItem);
```

Example 7-14: Clients can specify functions for the server to invoke.

---

[33] Following conventions you would "call" a function called "someFunc" from the client. The server has named this "SomeFunc" in order to follow C#'s naming convention.

Each method handles serialization behind the scenes using JSON. This means that you can send objects back and forth. You can also annotate properties to shorten property names (to reduce bandwidth usage). To get this back in readable form on the client, you have to write some manual code [107].

The serialization even handles date objects. I ran into a problem with this, as Date-objects deserialized as UTC time rather than GMT + 1. Problems with dates therefore persists, even though it seems like SignalR has solved it at first glance.

7.4.3 Maintainability

SignalR is just a real time layer within a web application, just as Socket.IO. This means that the only entities you have to test in addition, are the Hubs or the Persistent Connection classes. While I will focus on the Hubs, most of the principles I will discuss applies to Persistent Connections as well.

The Hub class, which is the abstract class all Hubs inherit from, is made with testability in mind. Testing a Hub class directly, is not feasible. This is because you have to mock the "Clients" object as this has dependencies you don't want in your test. Such a scenario is not uncommon when writing tests. What you do, is write a class that derives from the Hub you want to test and make mocks within it [106]. Setting up tests like this allows you to verify all logic within a hub. (Example 7-15).

```
[Test]
public void Echo_should_register_a_ReceivedAtServerEvent_in_monitor()
{
    _loadHub.Echo(_message);
    var expected = new[] {1};
    _monitor.ReceivedAtServerEvents.Values.ShouldAllBeEquivalentTo(expected);
}
```

Example 7-15: Unit test code for a SignalR Hub method.

The site I found information about this also claims that you can test that the Hub sends out correct values. I was not successful in doing this. This is not critical, as such functionality is more usually tested through functional tests.

7.4.4 Browser support

All major browsers are supported by the library. As with all the other frameworks, you can specify what transport to use. However, SignalR makes an annoying choice for you. If you provide "foreverFrame" (streaming) as transport in a browser that supports Server-Sent Events, you're not allowed [108].

7.4.5 Maturity

The development has gone on for a couple of years, but the library didn't reach 1.0 until early 2013 [109]. One of the benefits of being a Microsoft supported project, is that it has a strong team working with it. As a results, a 2.0 release surfaced within a year of 1.0 and both 1.x.x and 2.x.x versions receive updates frequently.

Another sign that Microsoft provides heavy support for SignalR is that it has become an integrated part of the ASP.NET framework. Newer versions of Visual Studio has project templates for it. The IDE itself actually uses SignalR for hot code push functionality [110]. Furthermore, a lot of attention goes into promoting the library at conferences. All this shows that Microsoft has confidence in the technology.

After the 1.0 release, I find it unlikely that the code platform and structure of SignalR will undergo any drastic changes. There may be additions, but overall, the current form of the library is very stable.

One sad thing about SignalR is that you must use IIS8 to harness its full potential. IIS8 is the only IIS version with support for WebSockets [57].

7.5 Meteor

Meteor is a web application framework with real time in focus. As of writing it has official versions for Linux and Mac. The unofficial Windows version is a little behind the official. During my work, this was not the case, and both were version 0.6.3. Meteor did not have support for MySql at this time, so I used the supported MongoDB instead. This means that I was not able to use the common functional test case described in section 6.6.3. Meteor also have integrated support for a client-side template language. The common user interface described in section 6.6.1 was thus not used.

7.5.1 *Documentation*

Even if Meteor is available for Mac and Linux only, the documentation mentions the unofficial Windows version. It is a little hidden, since you have to go via the "supported platforms" site [111]. The fact that Meteor only supports UNIX based operating systems may seem a little weird. But it is only natural, since Node didn't support Windows in the beginning [112]. In addition, maintaining only one type of operating system is easier, and makes the development process go faster.

The framework is a work in progress, and parts of it is changing all the time. This goes for the documentation as well, but I have the impression that updating it isn't the most prioritized task. Nonetheless, the constant changes makes it a little hard to follow sometimes. "Does this apply to my version?" was an often asked question. I knew that I would run into this problem when I chose Meteor, but it could have been more clear about what's set in stone and what's not. Node is also a work in progress, but their documentation does a thorough job at showing the status of different aspects. Meteor has some red text here and there, a feature that doesn't do the job as well as Node does (Figure 7-5).

From Node docs:

Domain

Stability: 2 - Unstable

From Meteor docs:

transform functions are not called reactively. If you want to add a dynamically changing attribute to an object, do it with a function that computes the value at the time it's called, not by computing the attribute at transform time.

In this release, Minimongo has some limitations:

* $pull in modifiers can only accept certain kinds of selectors.
* findAndModify, aggregate functions, and map/reduce aren't supported.

All of these will be addressed in a future release. For full Minimongo release notes, see packages/minimongo/NOTES in the repository.

Figure 7-5: Hard to keep track of stability of features in Meteor's documentation.

Meteor's documentation features many examples, but also quite a lot of text. This would be fine if it weren't for the design of the documentation page. All white and black makes things blend together, making it a little hard to read. Other than that it is well structured, showing a dynamic menu to the left that shows you where you are at a given time. Sometimes, this functionality breaks. But since you can navigate by clicking on any element in the menu, this isn't too much of an inconvenience.

Overall, the documentation is impressively detailed for something that isn't complete. Maintaining it requires a lot of work besides driving the project itself forward. There are also a few, simple samples and a few videos that show certain aspects of Meteor. The samples are well enough, but I found the videos hard to follow. They show a lot of code in a short amount of time, which isn't good for learning purposes. It leaves me to believe that they are more for promotional purposes. You can see one of the videos on Meteor's homepages [113].

7.5.2 Simplicity

Simplifying developing web applications is the motivation behind Meteor. Right now, it is not simple. But I can see that if the smart package system [50] works well, it will make it possible to write a lot of functionality fast. The concept is that you can build applications from a collection of packages. Then you write some code to wire it all up. This makes certain pieces of Meteor applications highly reusable.

With such a high level of abstraction, it feels like there is a lot of magic going on. Sometimes this is a little confusing. More than once I had to look twice at my code in my browser's debugger to make sure that it was what I wrote. Meteor handles bundling and wrapping of your code before it serves the client. This means that your code is always wrapped into a scope for you (Example 7-16). I must say that I prefer to handle this myself, as I think that it makes the code more readable and better structured.

The actual file:

```
1    addItem = function(item) {
2            if(!item.name || !item.minPrice || !item.expires || !item.addedBy) {
3                    return false;
4            }
```

After bundling:

```
1  (function(){ addItem = function(item) {
2      if(!item.name || !item.minPrice || !item.expires || !item.addedBy) {
3          return false;
4      }
```

Example 7-16: Meteor wraps your client side files automatically in a scope.

Meteor encourages the declaration of global level[34] functions and variables. By doing so, it isn't always clear what the effects will be. In the above example, the function, "addItem", ends up in the global scope. If this code was for a package, it would be global to the package only [50]. Most front-end developers would have second thoughts on this practice. General JavaScript development discourages the use of global variables.

Another thing that is a little tricky, because of Meteor's "magic", is structuring files with respect to dependencies. Files are served to the client in a specific order, based on how deep they are in the file structure and alphabetical. As a result, you sometimes have to make an extra folder just to ensure that one file loads before another. Meteor suggest using packages as a solution for this. I believe that this may solve the issue. Since support for making your own packages was limited when I tested Meteor, I was unable to test this.

---

[34] If you declare a variable without the keyword "var" in front of it in JavaScript, you declare it globally, even from within a scope. Outside a scope, it is global even with "var" in front [149].

Many of Meteor's features are revolutionary, but they also feel a little weird. For instance, publishing a record in the database, makes it available to all clients. The result is that any change to this data is broadcasted to all clients. As real time goes, this connection to a data set is unlike anything else. It somewhat resembles Lightstreamer's items (see section 7.2.2), but these items are not necessarily connected to a database.

Another typical real time feature is the ability to send a simple message from server to client or vice versa. With Meteor it is not possible to do so without involving a data set. In the test application, I implemented request/response functionality through Meteor methods (Example 7-17). For messaging from one client to another, this does not work.

```
Meteor.methods({
        login: function(username, password) {
                var user = verifyLogin(username, password);
                if(user) {
                        this.setUserId(user._id);
                        return user;
                } else {
                        throw new Meteor.Error(404, "Wrong credentials");
                }
        },
```

Example 7-17: Meteor method used for request/response functionality.

Having direct access to the database from both client and server is something that, as far as I know, no one has done before. It lets you write database queries on the client. Security is an obvious issue here, and Meteor has introduced a package for authentication. It is also encouraged to keep sensitive code on the server.

Clients emulate the "happy day" scenario of a database operation by default. As a result, you may see some changes starting to happen, before they snap back as the server responds [113]. From a user's perspective this is a little odd. The functionality can be overwritten, but in my opinion it should be off by default.

All in all Meteor offers many new features to web application development. It will be interesting to follow the project in the future. I think it can get a lot of users, but I don't see large corporations throwing out the more traditional frameworks to use Meteor instead.

7.5.3   *Maintainability*
As of now there is no official testing framework for Meteor. I find it a little worrying that the roadmap lists this as *"In 1.0 if time permits"* [74]. Testing is essential to keep maintainable code, and should be a part of any framework from the beginning. Especially with a framework with as many tight couplings as Meteor, where almost everything work together.

Still, there are ways to test an entire Meteor application. As of now, they feel a little unnatural, like you're hacking the framework to get it working. My approach was using Node with Mocha as test runner. Then I used a module called "unit-testling" [114] that allows you to inject mocks into another module (Example 7-18).

```
var testling = require('unit-testling'),
        should = require('should'),
        sinon = require('sinon'),
        mocks = require('./mocks.js');

mocks.mock.Template.stub('content');
var Session = mocks.mock.Session;

var contentTemplate = testling.load('../client/contentTemplate.js', mocks.mock);

describe('Template.content', function() {
        describe('#loggedIn() and #notLoggedIn()', function() {
                it('Should return visible when Session["logged_in"] is true. notLoggedIn should…
                        Session.set('logged_in', true);
                        contentTemplate.Template.content.loggedIn().should.equal('visible');
                        contentTemplate.Template.content.notLoggedIn().should.equal('invisible');
```

Example 7-18: Using "unit-testling" to inject mocks.

Meteor files are not Node modules. With some files, this was not an issue. But testing files with global functions was another issue. To make these tests work, I had to add some test specific code into the files I wanted to test with this method (Example 7-19).

```
if(typeof exports !== 'undefined') { //add testing capabilties
        exports.addItem = addItem;
        exports.placeBid = placeBid;
        exports.removeItem = removeItem;
        exports.usersBids = usersBids;
}
```

Example 7-19: Making a Meteor file testable as a Node module.

Testing Meteor is something that will benefit from integration testing of some form. Laika is a testing framework for Meteor that does just this [115]. Unfortunately, I was unable to get this working on Windows, but it looks promising. Laika proves that it is possible to write tests for Meteor in a simple manner. Whenever an official testing framework surfaces, it will probably resemble Laika in a lot of ways.

As Meteor shares a lot of code between the server and the client, a new problem occurs. Should you test the code on the server, the client or both? And how do you test code that triggers a database update on the server from the client? There are many unanswered questions with Meteor as of now. I believe that the answers will come soon and that they will be satisfactory. Many developers have fate in the project[35], something they wouldn't if it didn't look promising.

7.5.4 *Browser support*

Meteor supports all major browsers. I had some issues regarding the WebSocket support of SockJS. While it does handle graceful fallback, it refused to use WebSockets in Chrome when the address was localhost. This turned out to be a bug with SockJS [116], and pointing the browser to 127.0.0.1 solved the problem.

7.5.5 *Maturity*

The development of the project has been steady since 2011. Recently, it has started to receive a lot of appraisal from developers, thus boosting the community. In 2012 it received a substantial financial contribution [117]. This assures that the team can work on Meteor full time rather than besides other work.

Meteor introduces a new way of thinking–there is no other web application framework like it. Real time features becomes more popular every day, and Meteor puts this in the center of its application-model. A repercussion of this, is that the framework will not be suitable for every task. Then again, what framework is? The developers promises to solve a lot of problems and revolutionize web applications. Only time will tell if they offer too much. But if they deliver, I think Meteor will become a popular choice for small to medium sized projects.

Currently, Meteor is far from ready to be used in production code. Drastic changes to the API may still occur, and there are no guarantees offered by the documentation regarding any aspect of the framework. This means that the current state of the project is only suitable for case studies and hobby projects. With so many innovations, the framework will also need time to prove itself after it reaches a stable version. Having database access from the clients is the aspect that really needs to prove itself. If it turns out to be secure, and if they can support all major databases, it will no doubt change the way we think of front-end forever.

---

[35] A quick Google search reveals a lot of articles and blogs about Meteor. This indicates that Meteor has the attention of many people in the web development community.

Project part 2:
Load testing

8 Methodology

This part will focus on the last two problem statements (see section 1.1). That means it will be about performance. Both for the individual frameworks and the different transport mechanisms. I will strive to find out what frameworks has the best performance. The question of WebSockets versus HTTP will also get a lot of focus.

8.1 Test scenario

Real time applications are about distributing updates at once to multiple clients at the same time. The scenario I designed follows this concept. For each test, a set number of clients follow a strict message flow, recording data along the duration of the test. One of the clients is dubbed "master" and handles starting a test. Figure 8-1 describes the flow of a test. Section 8.2 provides further insight into this.
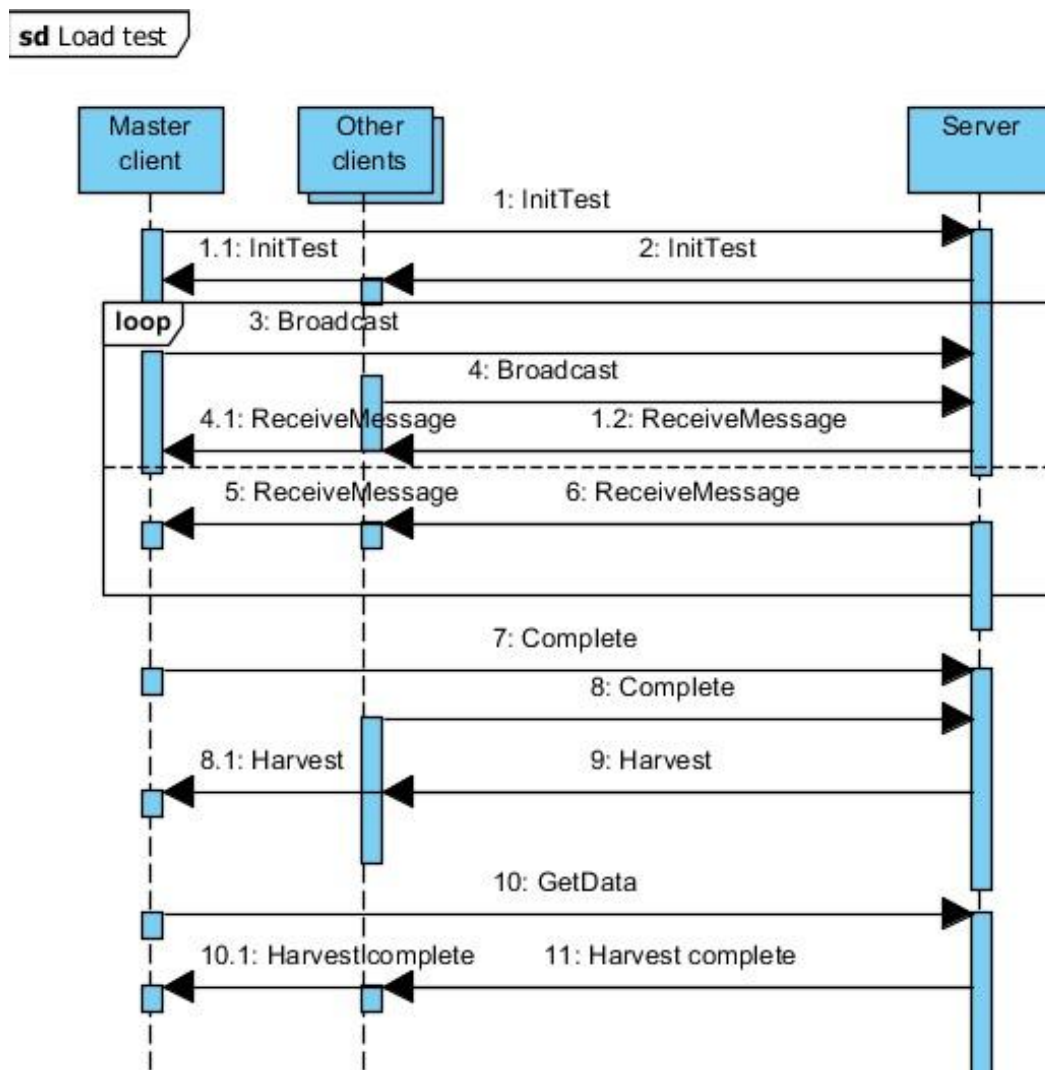
Figure 8-1: Message flow in a test run.

Along with the "initTest" message, the master client sends information about:

- What test to run. The "echo" option starts a test where each message received by the server, is sent back to the caller. The "broadcast" options starts a test where each message received by the server, is sent to all clients. I ended up just using the "broadcast" option.
- How many clients are connected. Instead of letting the server count this, I find it easier to just tell it.
- The spacing of the x-axis in the graphs generated by the test. Numbers on the axis represent time intervals in seconds. For instance, the value "0", represents the interval from zero to one second if spacing is set to one. I ended up running short tests, so this value was set to one for all tests.
- The start time as recorded client side. To eliminate time differences, the server records the start time as well. Calculations for client data use the first, while server data uses the latter.

All messages from the "complete" message and onwards handle exchange of data. At the end of a test, clients has access to an object with the complete dataset of the test. This allows me to separate calculations and displaying of the results from the tests themselves.

In the previous part of this thesis (see page 33), all communication involved a database. For the load tests, this is not the case. There was no database involved in these tests.

8.2 Test data

While the tests run, both the clients and the server collect different data. Message frequency data is the server's responsibility. When a client sends a message to the server, it gives it a timestamp. The server uses this to calculate how many messages all clients sent during a given interval. It also registers how many messages it received and sent during the same interval. One goal of recording this data is to find out if any messages are dropped. The other is to determine if the server manages to keep up with the message stream.

Messages received by the server are registered immediately after receiving a message. The registration of messages sent from the server, happens as close to the send event itself as possible.

The other telemetry measured automatically by the tests is latency data. Each client needs to calculate the latency of its own messages. To achieve this, a message gets an unique ID. This ID is a string built after the following format: *"c:<clientID>m:<number of messages sent by this client>"*. For example: the 37th message of client number 13 has the ID "c:13m:37".

A client registers that it has received a message only once, and calculates the latency using the timestamp for when it was sent. There is one drawback to this technique: I have no guarantee that a framework sends a message to the caller first when it broadcasts. This introduces some insecurity in the values, but not much. The extra latency will never exceed the time it takes to send a message to all clients.

In addition to the data described in the previous paragraphs, I monitored some telemetry with other tools. What a certain framework or transport required of the server in terms of memory and processor was measured. Network usage was the data I expected to see the clearest differences between transports, along with latency. Hence, it was important to measure how many bytes got sent during a test.

8.3 Test setup

There are several ways to implement the test scenario:

- Using an existing tool.
- Using console applications as clients.
- Using headless browsers[36].
- Using real browsers.

I have not found any tool that can handle testing real time applications in the manner I want. The first alternative was therefore not suitable for this thesis.

Most load test setups I have seen use some sort of desktop or console application to emulate clients. Some examples of this are Crank [118], Apache JMeter [119] and Gatling [120]. This test however, has many aspects that I have yet to see in any other scenario.

I tested five different frameworks. All used WebSockets, but I also tested the different fallback transports. Each framework expects messages in a different format, and they all have different ways of connecting a client to the server. Achieving this with console applications would need a lot of overhead to support it all. WebSockets would need different code than HTTP as it is a separate protocol. Furthermore, I would have to manually construct each message to fit the format of a given framework.

A better solution is to make use of the JavaScript clients each framework provides. Using headless browsers, I could still use a console application. But rather than acting as clients, this application would just launch a given number of headless browsers. Sadly, few viable headless browsers exist for my purpose.

---

[36] A headless browser is a browser without the graphical user interface [150].

Phantom is the most widespread, but the current version (1.9) does not support the newest version of WebSockets [121]. Phantom uses Google WebKit. Slimer is the Gecko (Mozilla) counterpart. This does support WebSockets, but it is an immature piece of technology. Also, it is not completely headless yet [122]. I also considered HtmlUnit [123], but its WebSockets support seems somewhat outdated. The changelog shows that it got support for it in 2012, but it is not mentioned since [124].

The final option is to use real browsers and have multiple clients in each open window. This also allows for use of the JavaScript clients each framework provides. But it is a solution that demands more resources of the client machine. In the end, it will not be possible to have as many clients with this solution as with the others.

8.4 Choice of setup

I chose to go with real browsers. Writing many different console applications could easily have become a time trap, resulting in uncompleted tests. Using headless browsers would have been the optimal solution, but I find Slimer, the best candidate, to be too immature. Any bugs with the testing software could have given false results.

To start up the browsers I wrote a simple Selenium application that I named "ClientLauncher" [13]. Firefox turned out to be the ideal browser as Selenium has good support for it. It was also the browser that handled several clients without any problems.

I used 60 clients running in 30 browser instances. Message frequency was set to two messages per second per client. Message method was set to "broadcast", resulting in a message frequency from the server to the clients of 7200 messages per second. The load this produced is not substantial, but it was enough to generate differences between both frameworks and transports.

The clients were hosted on my own Asus K55V laptop. It has the following specs:

- Intel Core i7-3610QM CPU with 2.30 GHz.
- Eight Gigabytes of RAM.
- 64 bit Windows 7 Home Premium.

To host the servers, I used a desktop computer from HP with the following specs:

- Intel Core 2 Duo-T7600 CPU with 2.33 GHz.
- Eight Gigabytes of RAM.
- 64 bit Windows Server 2012 R2 Standard Edition.

During each run, the two computers was connected to each other using an Ethernet cable.

One of the major benefits of using browsers is that it allows for a lot of shared code on the client. Eight JavaScript files make up the client-side code. Of these, seven are shared between every framework. The file "Socket.js" handles communication with the server and had to be rewritten for each framework. There is a total of 50 tests for the three main files that handle the tests. This ensures that the client works as specified.

The other JavaScript files handle things like setting up a global namespace and input events. I could easily verify manually that these aspects were handled correctly. For this reason I did not write unit tests for these files.

Each of the frameworks needed different servers, but I wanted them to follow a strict implementation. There are two main entities for registering data: the "loadhub"[37] and the "monitor". 19 and 16 tests respectively ensures that these entities perform to specification. In addition, most frameworks needed some extra code to wrap these entities together with sending messages. Play and Lightstreamer, for instance, needed to serialize data to JSON. They share the class "JSONHelper.java" for this purpose (also unit tested).

8.5 Number of runs

To get viable results, I did several runs and calculated the average values afterwards. The results turned out to be mostly stable from run to run. Therefore, I decided that ten runs of each combination of framework and transport would suffice.

8.6 Displaying data

I chose to use Highcharts [125] to display charts with all collected data. This is a simple and powerful API that provides graphs that suited my need. To format the data gathered by the clients and servers, I designed a simple Web API [13]. After a test is done, the master client is in possession of all the raw data. It sends this to the Web API. The API processes the data and returns three objects that the Highcharts API can make use of. A total of 33 unit tests ensures that also this part of the setup performs as expected.

While the Web API provides charts for each run, it is the average values of several runs I need. To help with this, I designed a "Chart Merger" application [13]. This contains a series of (unit tested) functions to extract the average values of different types of tests. It then presents this in Highcharts graphs. The Chart Merger also handled the data I gathered manually. All graphs presented in chapter 9 and 10 was generated using the Chart Merger.

---

[37] I implemented the setup with SignalR first. I therefore chose to use its terminology.

Some of the graphs are made from special combinations of the raw data. An example of such a graph is one with data from multiple transports using a single framework. All raw data files in the folder "Loadtests/Results/Custom" [13], except one, are files that generate graphs like this. To for example display a graph from a file with data from SignalR using both WebSockets and Server-Sent Events, you need to input "SignalR SSE" in the "Add custom frameworks" text-field. Other framework names can be written here using a comma-separated list. These have to correspond with names in the selected JSON file (for browser data).

One exception is the file containing data to display data for messages sent by the server in the graph representing both messages received by the server and messages sent by clients. I did this in section 9.3. To reproduce this graph a single line of code in the Chart Merger application has to be altered. On line 108 in "merger.js", you have to add a string in the array to match the series-name: "Messages sent from server / 60". I named this custom series this since it represents the number of messages sent from the server divided by 60.

A final note on graph generation is that the first graph you see, in the Chart Merger, has the title "Messages sent from clients and received by server pr. second". However, in this document you will see only a few graphs with this title. I have not changed any code to change the title of the other graphs. Instead, a simple image editor was used to tailor the headings to fit my needs.

8.7 Configurations

With SignalR, IIS benefits from a few configurations. I followed the steps given by the wiki of SignalR's GitHub page [126].

Lightstreamer required some more work before it was ready for testing. First of all, the free version limits the number of updates to only one per second [87]. Hence, I had to upgrade. Weswit proved very helpful, giving me free access to the most expensive version, the Vivace edition[87].

My initial results showed that Lightstreamer was behind the other frameworks, so I turned to Weswit for help. Appendix A shows the whole e-mail correspondence regarding server configuration. One of the critical things I learned from them is that the JVM needs to warm up before load tests. As a result, tests done with Play and Lightstreamer ran for 30 seconds before I started recording data. A test of four minutes showed that the data values stabilized around this mark. Hence, there was no need for a longer run.

Weswit also helped with a few settings needed for the Lightstreamer server. Not all of these setting made a difference for me, so I didn't use all they suggested. The only tweak I made was setting the attribute "max_delay_millis" to 0.

8.8 Monitoring network traffic

Several tools were tested to find one that fit my needs. I was interested in a tool that could show the total amount of bytes sent given a certain filter. It also needed to have functionality to inspect packages for both WebSockets and HTTP traffic. Network Monitor [127], Wireshark [128] and Fiddler [129] were the most prominent that I considered. In the end, the choice fell upon Wireshark as it provided exactly what I needed.

During each test, I captured packages using the following filter: *"((ip.dst == 192.168.137.61 and ip.src == 192.168.137.1) or (ip.dst == 192.168.137.1 and ip.src == 192.168.137.61)) and (websocket or http or tcp)"*. This means that I wanted to see packages either going from the client machine to the server or the other way. Furthermore, I only wanted WebSockets, HTTP and TCP traffic.

8.9 Monitoring of processor

While I could have chosen an advanced tool with pin point accuracy, this was not what I wanted with this data. The most interesting for me was to see if there were clear differences between different transports and frameworks. A simple tool that could have been used is Perfmon [130]. However, this tool offers counters for processor time, not for the total usage. The server is the main program running on the test machine. The counter for processor time shows how many percent of the total used percentage a process uses. Because of this, the server would always show close to 100% usage[38]. Windows Server 2012's Resource Monitor was therefore chosen for this purpose.

8.10 Monitoring of memory usage

Memory usage falls into the same category as processor usage. I wanted to see if there were any clear differences. The Task Manager in Windows was more than enough of a tool for this job. Readings were taken at the end of each test, as this proved to be the maximum usage during the test runs.

8.11 Different servers and platforms

The frameworks run on quite different servers and platforms. Node is for instance a lot more lightweight than Java or C#. This is because Google's V8 engine, which it builds on, is implemented with C++ [131]. C++ compiles to machine code, and it does not need a runtime to function. Both Java and C#-code compiles to bytecode that has to be interpreted by a runtime. For Java this is the JVM while C# has the CLR. As a result, C++ is a faster language than both C# and Java [132].

---

[38] If one program is running, it is the only program using the processor. Thus it uses 100% of the processor's time.

I chose a coarse grained method for monitoring resource usage of the different frameworks. Therefore, the results for processor- and memory usage shows more than just the frameworks' usage. A substantial part will belong to the server in some cases.

This method allows me to see what the different servers require of the machines they run on. In my opinion, this data is more relevant than what the frameworks themselves use. The server is a part of the total package, and I believe that it is interesting to see how these perform compared with each other.

No matter what method I used to measure machine resources, I still got an overview of how each transport performed. This comparison couldn't have been done across multiple frameworks anyways. For instance, it would be wrong to say that long-polling is better than WebSockets based on Socket.IO with long-polling and SockJS with WebSockets. This is because the two libraries have different implementations.

8.12 Use case of test setup

I should point out that my setup is tailored to suit the needs of this thesis. It is the web application aspect of each framework I am testing. The use of JavaScript clients is thus the most natural. Results should be read with this in mind. Some of the frameworks support clients on other platforms than browsers, and have other use cases than web applications. These frameworks may prove to perform differently under such circumstances.

8.13 Limitations

This section describes various limitations I ran into using the chosen test setup.

8.13.1 Meteor

Meteor has the real time component tightly embedded in its core. This made it impossible to use more than one client pr. browser without fiddling around with Meteor's source code. As this easily could have broken certain aspects of the framework, I decided to test Meteor's real time component alone. SockJS therefore replaces Meteor in this part of the thesis.

8.13.2 Using browsers

There are certain drawbacks to using browsers as clients. Most obvious is the fact that you cannot have more than a few open connections at the same time. The HTTP/1.1 standard states that no client should have more than two connections to a single host [18].

Nonetheless, most modern browsers have increased this limit to somewhere between four and eight [133]. Consequences of this are that I cannot have more than two or three clients per browser, and I cannot have a high message frequency. If I do, I risk reaching the maximum number of connections, thus introducing extra latency.

Another repercussion is that the number of clients will be limited by the client machine. An idle browser takes up about 100 megabytes of RAM on my machine. Considering that each browser will store some data as well, this number grew close to 200 megabytes. My machine was pushed to the limit with 30 browsers running.

8.13.3 Network capture

I started a new capture when I started the tests. This turned out to have huge drawbacks that I did not discover until all tests were run. Open WebSocket connections was not recorded correctly. Messages going from the server to the clients are there, but they show up as TCP traffic. The other way is masked, so there is no way of knowing if the data is correct or not.

HTTP-streaming and Server-Sent Events suffered from similar problems. It also seems to be some issues with long-polling with SignalR. Some messages that I expect to be in the capture are not there (cursor messages). This leads me to believe that the captured data is not accurate enough to be presented as results. It can give an indication, but that is all.

To present some more believable results, I have done a calculation of the theoretical throughput of each test type. The basis for each calculation is a capture with one client sending ten broadcast messages over two seconds. For these tests, I started the capture before I navigated to the test page, thus ensuring that all packets were recorded.

There are obviously sources of error with this approach as well. Some of the frameworks may compress data more or send multiple messages in one package during higher loads. Therefore, I must stress that the results from my calculations do not take this into consideration. The actual performance of each framework may be better than what the calculations indicate.

A benefit of doing this is that it allowed me to show results for transports that are not supported by the frameworks in Firefox. This applies to HTTP-streaming with SignalR and Server-Sent Events with SockJS. Keep in mind that there may be some differences between how browsers handle different transport mechanisms. Still, it gives an indication.

The basis for the calculated results is on GitHub [134].

8.13.4 Streaming with Play Framework

Streaming with Play Framework required two forever-frames in each browser. I have not been able to understand why, but this did not work. Even when I used one client per browser, connections failed after six. For this reason, I excluded HTTP-streaming with Play from the tests. However, I was able to get a basis for calculations of network traffic for it.

8.14 Testing idle connections' resource usage

To find out whether WebSockets has potential to set a new standard for client/server communication, I measured how much resources idle WebSockets connections use. If it is to have any chance of replacing HTTP as transport method in static web pages, it should use close to no resources. There is no such thing as an idle HTTP connection in such cases, as the client requests content and gets it via a response. The connection is then closed. Using WebSockets for such a case would require the connection to remain open.

I performed two tests with each framework to look into this. One connected 1000 clients evenly spread across 10 browsers to the server. The other connected 4500 clients. This used 25 browsers with 180 clients in each. 4500 clients may seem like an odd number, but there are some reasons for it. I ran some tests, to see how many clients I could connect in a single browser. This was stable up to around 200 using Socket.IO. After this, some connections were refused. With a safety margin of 20, I ended up with 180 clients pr. browser.

My computer was able to handle about 30 instances of Firefox. Since these test would use three times as many clients per browser (60 in the other load tests), I wanted to add a safety margin here as well. To be safe, I landed on 25 browsers.

8.15 Message exchange with Lightstreamer

I mentioned in section 7.2.2 that the usage of a single "message" DOM element might degrade the performance of Lightstreamer. In the load tests I still used this approach. This is because I wanted the tests to be as equal as possible. Using Lightstreamer's subscription modes in a better way, may improve how it performs. Especially at higher loads.

Another reason for this decision is that I have more than one client in each browser. That means that most of the fields in the message gets updated for every new message that arrives. Therefore, the gain of having the ability to update single fields becomes almost neglectable.

8.16 Raw data

All raw data, except the network captures, can be viewed on GitHub [13] in the folder called "Loadtests/Results". Due to the size of the network captures, these cannot be uploaded to GitHub. If the reader wants access to these, send me an email at kris-90@live.no or kjohann@ifi.uio.no.

9   Results

This chapter describes the results of the load tests. Each graph shows the average values of the ten test runs for each framework/transport combination. For the results regarding messages, I only show a selection of graphs. The graphs I show, highlight some general trends in the results as well as anomalies I experienced. All standard graphs[39] generated by the raw data can be viewed in Appendix B. An analysis of the results is given in the next chapter.

The bar charts presented in this thesis have some values that may appear to be 0. This is not the case. That there is no bar means that the particular framework does not offer the particular transport.

Chapter 10 gives further insight into the data presented in the graphs below. There I use tables to show the highest and lowest recorded values as well as the average.

9.1 Messages sent from clients

With 60 clients and two messages pr. second, I expected a stable graph with the value 120 in the interval 0 to 14 seconds for a perfect run. A lot of the runs show this behavior, but a lot more than expected have certain anomalies. Figure 9-1 shows the results for WebSockets. It displays one example of such anomalies. Some runs went one second past the expected 15 second[40] duration of the test, resulting in a sudden drop during this extra second.

Figure 9-1: Messages sent from clients using WebSockets.

---

[39] A standard graph is a graph generated from the raw data in the folder "Loadtests/Results/High throughput".

[40] All data past the value "14" on the x-axis is after 15 seconds since 14 represents the interval from 14 to 15 seconds.

Project part 2: Load testing

Some tests had a lot more overtime than just one second. The graph below (Figure 9-2) shows results from runs using polling. As you can see, the results from Socket.IO stand out from the rest with as much as ten seconds overtime. This behavior was consistent through all the test runs. The Lightstreamer results were also consistent.

Figure 9-2: Messages sent from clients using polling.

The results from HTTP-streaming show a lot of overtime for Lightstreamer (Figure 9-3). This reflects only one run that lasted seven seconds too long. None of the other runs replicated this effect, but they still ran one or two seconds past the 15 expected seconds.

Figure 9-3: Messages sent from clients using HTTP-streaming.

9.2 Messages received by server

I expected the server to receive the same amount of messages that were sent from the clients in a given interval. The results show that this is the case for most of the tests. The graph below (Figure 9-4), shows both messages sent and received for all frameworks using long-polling. As you can see, the results are so close that there is almost one line in the graph. This is the expected behavior of a perfect run.
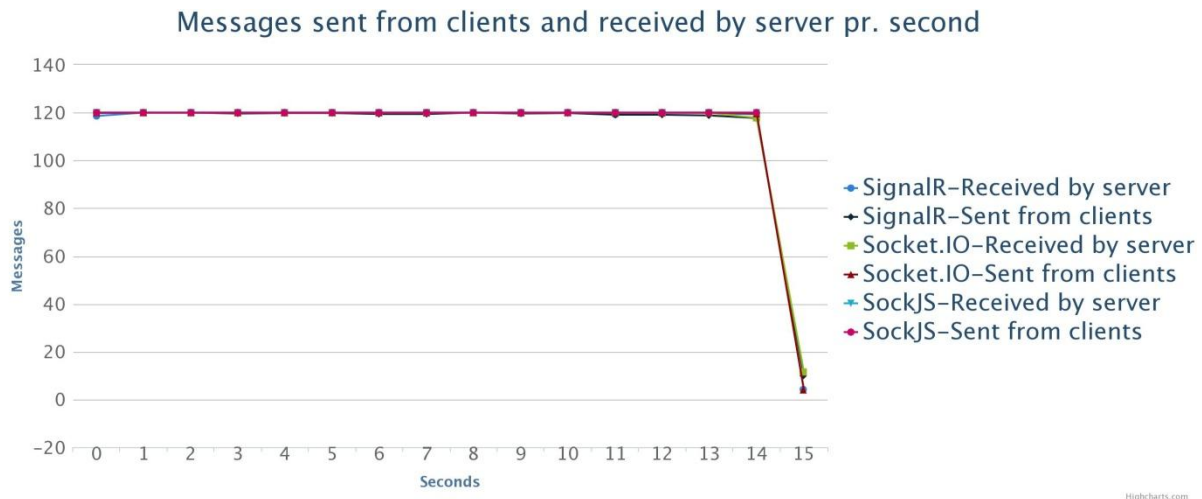
Figure 9-4: Messages sent from clients and received by server using long-polling.

Again, there are some anomalies. Figure 9-5 shows the WebSockets results for Lightstreamer. There are some deviations between messages sent and messages received. The difference is not significant, but it is enough to show that something is off. I will discuss possible reasons for such variations in the Analysis chapter.
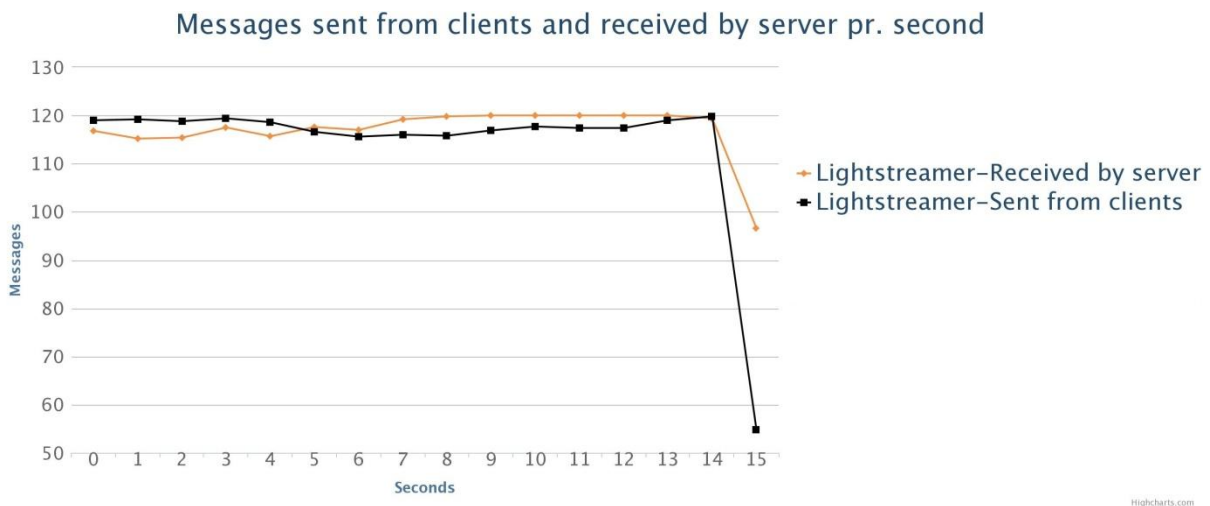
Figure 9-5: Messages sent from clients and received by server with Lightstreamer WebSockets.

While most of the variations are small, polling is the largest source of anomalies. The clearest example of this is Lightstreamer (Figure 9-5). It seems like the server worked in intervals, with peaks of more than 130 and bottoms of 105 received messages. These are variations of up to 12,5%[41], which is quite significant.

9.3 Messages sent from server

A perfect test should show that the number of messages received multiplied by 60 is a perfect match to the number of messages sent. It broadcasts every message to the 60 clients, thus the multiplication by 60. The following graph (Figure 9-6) shows this behavior for SignalR using Server-Sent Events. It is a small deviation the first second, but otherwise, the lines are almost 100% matches. This behavior was consistent throughout all the tests for all frameworks.
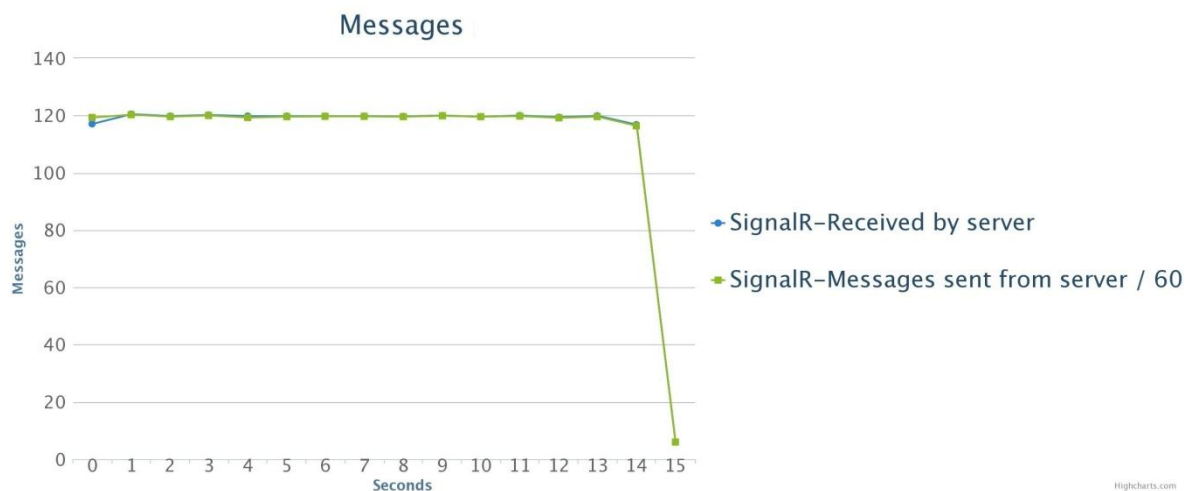
Figure 9-6: Messages sent by server (divided by 60), corresponds with messages received.

---

[41] I expected 120 messages, but 130 is 15 more while 105 is 15 less. A variation of 15 / 120 = 12,5%.

9.4 Average latency

This section will present the results for each transport. An analysis of the effects of a transport on latency will be given in the next chapter.

I did not warm up the servers for SignalR, Socket.IO and SockJS before any of the test runs. Looking at the results in this section, it seems that they may have benefitted from this after all. The most accurate results are therefore the ones recorded from five seconds and onwards.

9.4.1 WebSockets
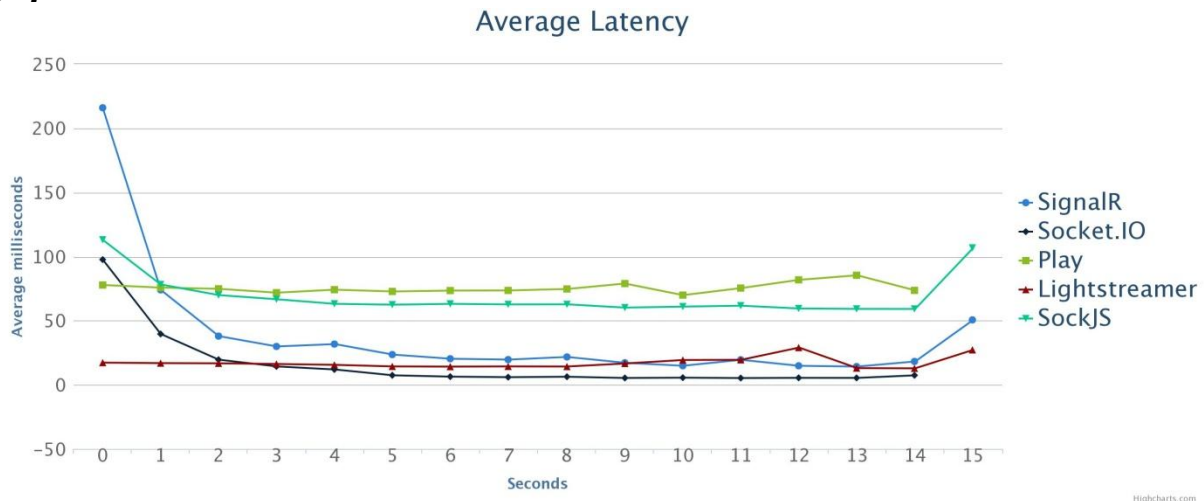
Figure 9-7: Average latency using WebSockets.

Both the Play- and SockJS implementations rely on some manual client handling. Broadcasting messages with these frameworks relies on a loop construct. The other three have internal mechanisms that handle this. Figure 9-7 shows a clear divide between the Play and SockJS and the other three. Of the best three, Socket.IO had the best performance as it stabilized around five milliseconds. Lightstreamer and SignalR follow each other closely, but did not stabilize in the same manner. On average, they had a latency of a little less than 20 milliseconds, with some peaks going above 25.

9.4.2   Server-Sent Events

Only SignalR provides Server-Sent Events as one of the transports[42]. The graph below (Figure 9-8) compares the results of this transport to SignalR's results using WebSockets. This is a very interesting case as it shows that Server-Sent Events matches WebSockets when it comes to latency.
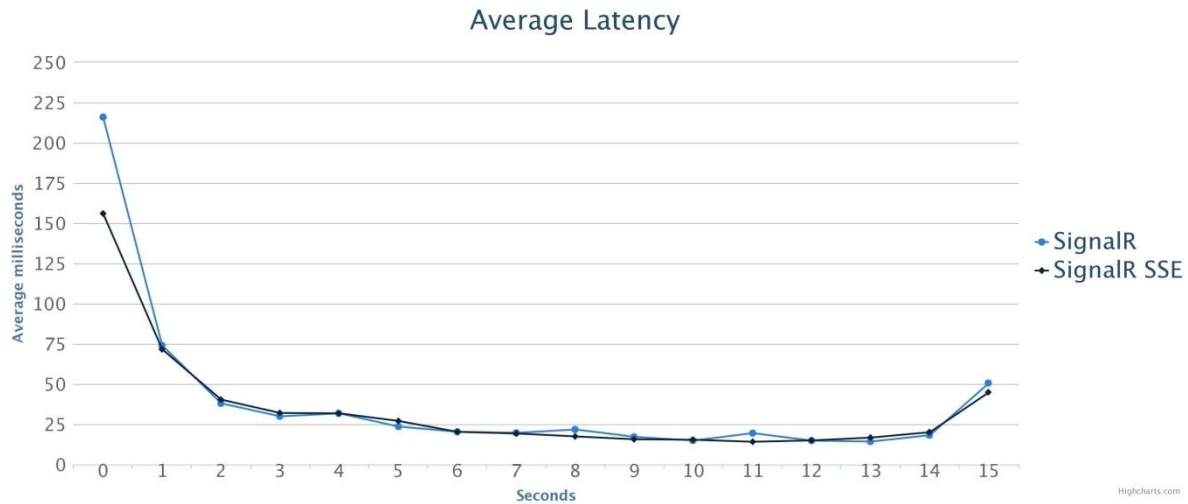
Figure 9-8: Average latency of SignalR with Server-Sent Events and WebSockets.

9.4.3   Http-streaming

In section 9.1, I mentioned that only one run with Lightstreamer had seven seconds of overtime. The graph shown in Figure 9-9 reflects this fact, as the latency increases a lot towards the end. As the other runs never passed two seconds of overtime, the data from 18 to 21 seconds should be disregarded. Prior to the 18 second mark, Lightstreamer performs well with HTTP-streaming. The latency is higher than WebSockets, but not as much as one might expect. SockJS displays the same behavior.

---

[42] With Firefox this is the case. SockJS has support for Server-Sent Events with Opera [48]

Project part 2: Load testing
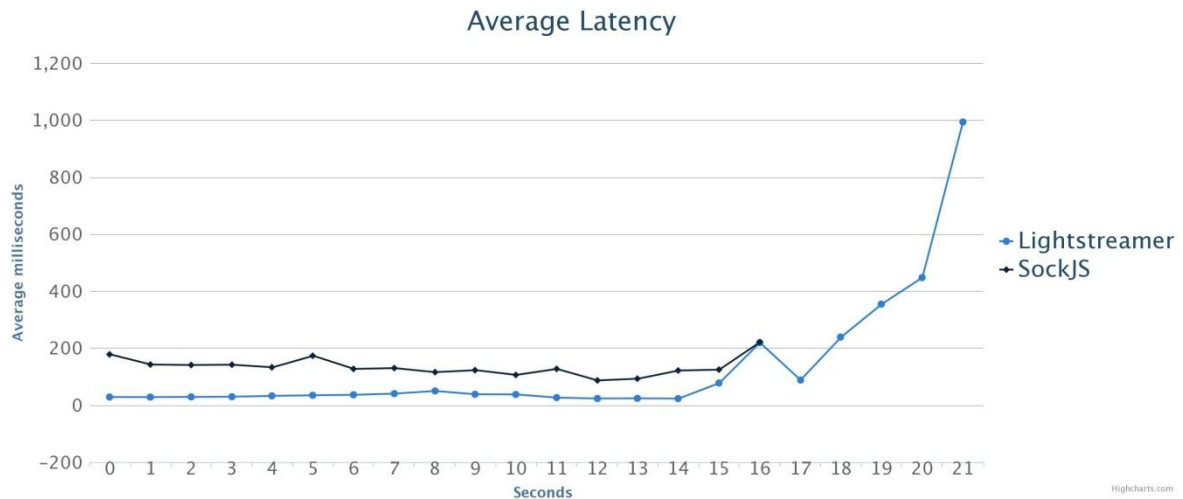
Figure 9-9: Average latency with HTTP-streaming

9.4.4 Long-Polling

As with WebSockets, Socket.IO outperforms the other frameworks with long-polling. However, as the graph shows (Figure 9-10), SignalR's latency decreases throughout the duration of the test. As discussed in the introduction to this section, I did not warm up the server for SignalR. The other transports with SignalR showed a stabilization after five seconds. With long-polling, this is not the case, even though it seems to be stabilizing a little around the ten seconds mark.
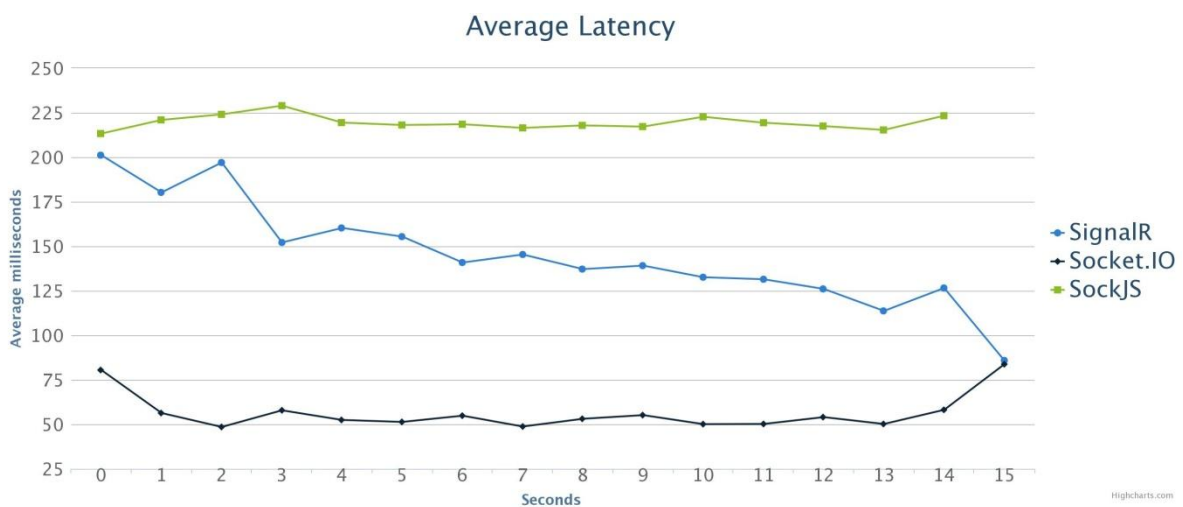
Figure 9-10: Average latency with long-polling.

9.4.5 *Polling*

Once again, polling is the greatest source of deviating results. As the graph (Figure 9-11) shows, Lightstreamer has a dramatic increase in latency towards the end, even with WS-polling. Socket.IO also has a peak with twice as much latency as the rest of the test. In the stable part of the graph, we see that Socket.IO does not handle polling well.
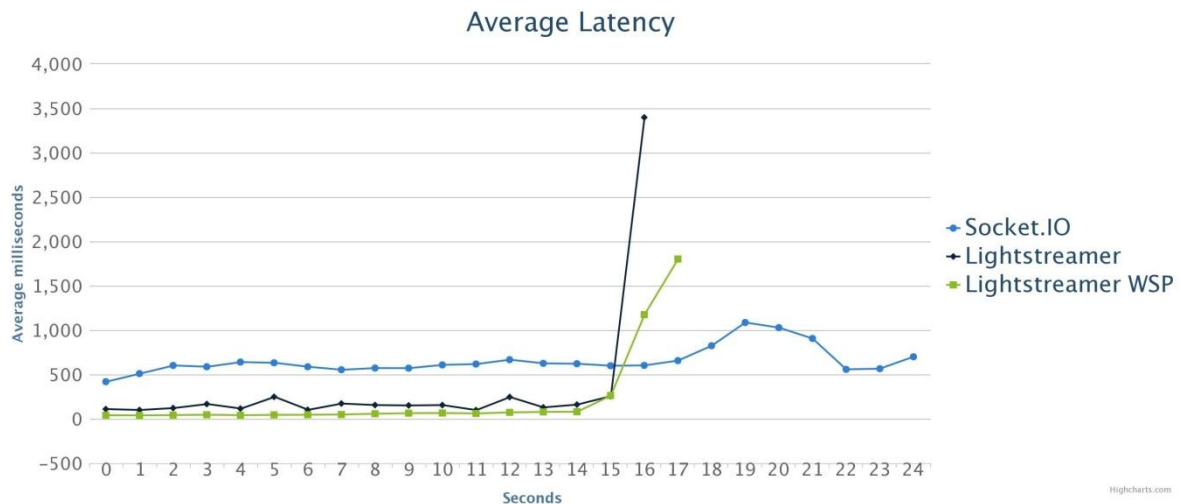
Figure 9-11: Average latency with polling.

Polling using WebSockets is more efficient than using HTTP by about 100 milliseconds. This is not surprising, but I still don't see the use case for this technique. As a browser has to support WebSockets to use it, it should be compared to streaming using WebSockets. Figure (Figure 9-12) compares the two, and it is obvious that WebSockets perform best when streaming data. Polling over WebSockets degrades performance more than three times on average.
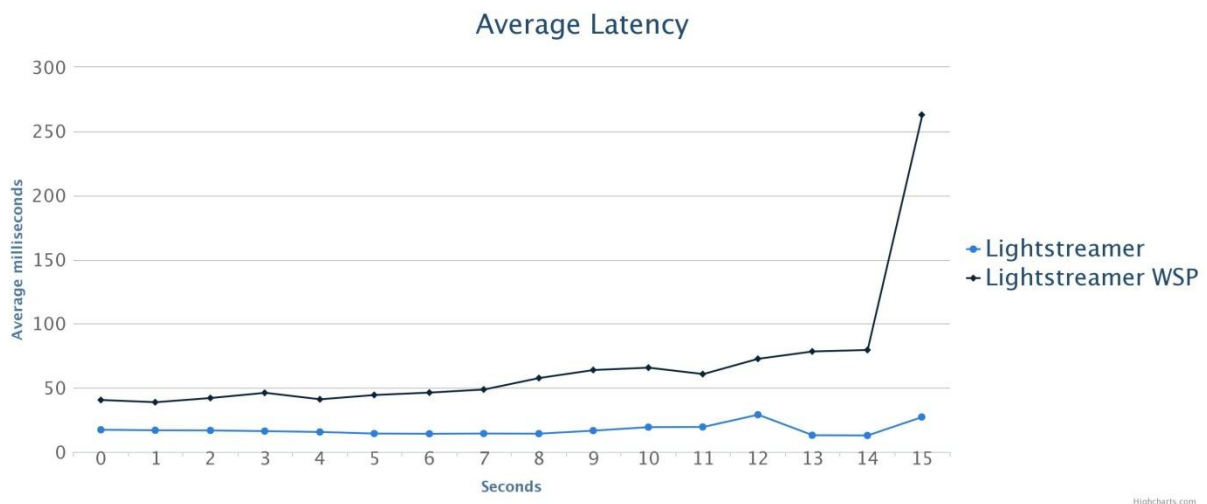
Figure 9-12: Average latency with Lightstreamer using WebSockets and polling over WebSockets.

9.5 Median processor usage

As discussed in the methodology chapter (see chapter 8), these result reflect the usage of the servers, not just the frameworks. The lightweight nature of Node shines through as both Socket.IO and SockJS takes up about half as much as any other framework (Figure 9-13). Otherwise, it seems that the Java solutions perform a little better than SignalR.
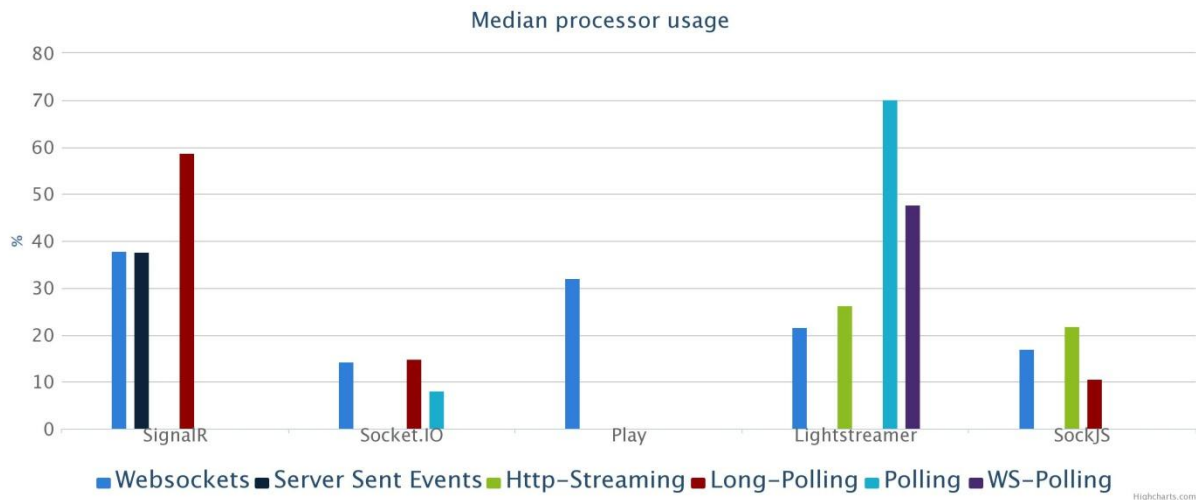
Figure 9-13: Median processor usage - all frameworks and transports.

9.6 Maximum memory usage

Again, the results reflect the resource usage of the servers as well as the frameworks. As with processor usage, one can see that Node outperforms both Java and C# (Figure 9-14). Otherwise, we see that SignalR, Socket.IO and SockJS have almost the same memory usage regardless of transport. Polling with Lightstreamer, stands out from everything else. It uses about nine times as much as WebSockets.

There is one source of error in the results. I measured the memory consumption at the end of the test. While I did not see any drops during the tests, there was some drops that occurred just after the test finished. Some of the raw data stands out from the rest because of this.
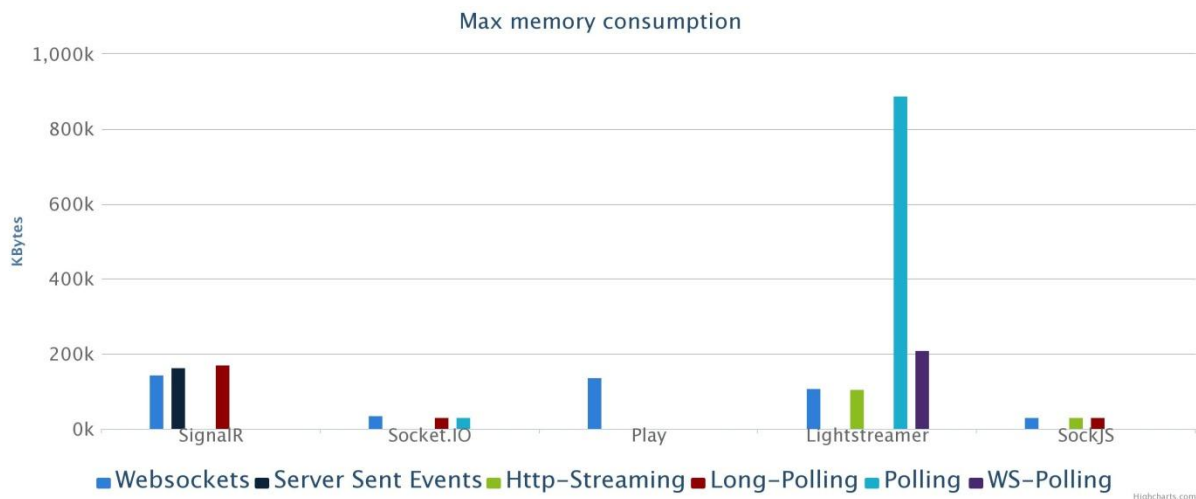
Figure 9-14: Maximum memory consumption - all frameworks and transports.

9.7 Bytes sent/received

In section 8.13.3, I mentioned an error regarding my usage of Wireshark to monitor network traffic. I still want to show the recorded data, as there are some implications in it when compared to the calculated network traffic. The following graph (Figure 9-15) is the result of the recorded network traffic.
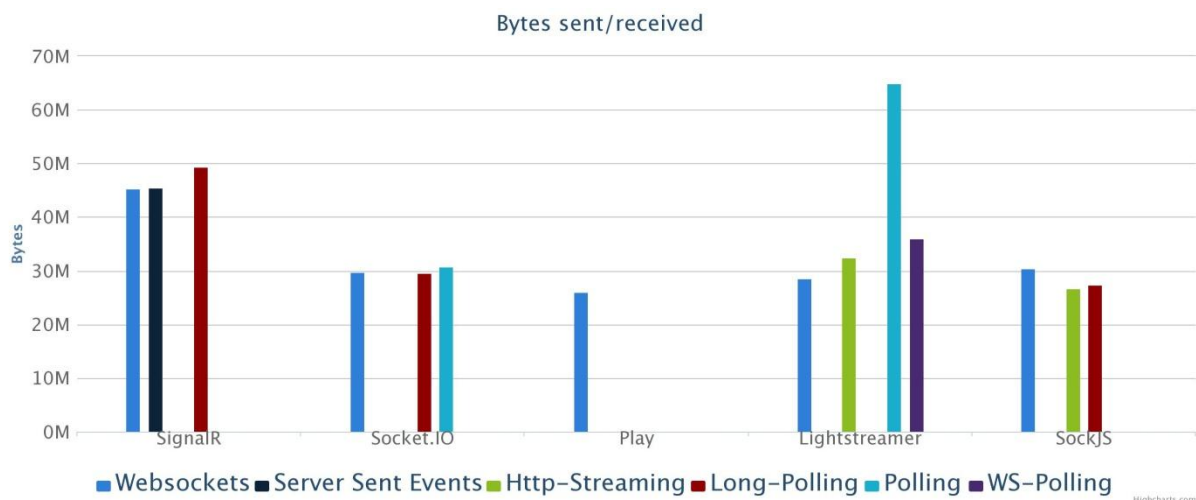
Figure 9-15: Bytes sent/received (captured) - all frameworks and transports.

The next graph is the data I calculated. (Figure 9-16). It shows that SignalR sends more data than any of the other frameworks. It also gives a clear indication that the streaming techniques use a lot less network traffic than polling and long-polling. There is one result that stands out from the general trend. Server-Sent Events with SockJS uses more than twice that of WebSockets and HTTP-streaming.
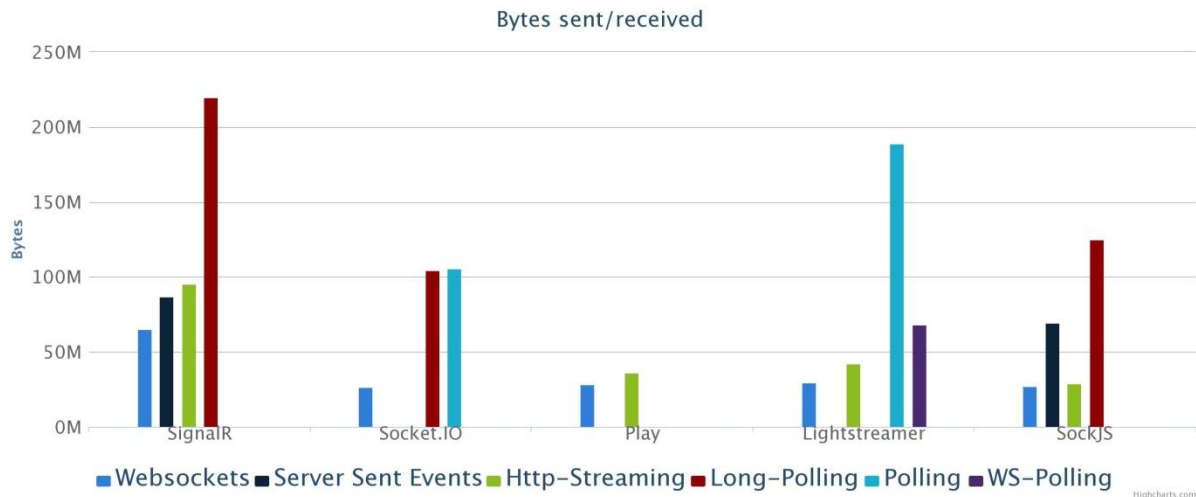
Figure 9-16: sent/received (calculated) - all frameworks and transports.

I expected Play framework to be the best framework in this category. It is the framework with the closest to a "raw" WebSockets implementation. But the results show that both the Node frameworks perform a little better. Lightstreamer is also not far behind.

9.8 Idle clients with WebSockets

Table 9-1 shows how much memory each framework used with 1000 and 4500 clients. Processor usage was mostly zero for all. There were some small peaks of one or two percent, but these can be ignored. The true processor usage was therefore nothing for all frameworks regardless of how many clients were connected. Because of this, I will not present any results about processor usage for idle connections.

Framework	1000 clients	4500 clients	Increase factor
Play	118932KB	223812KB	1,88
SignalR	188892KB	537064KB	2,84
Socket.IO	42724KB	81780KB	1,91
SockJS	38800KB	85996KB	2,2
Lightstreamer	166,788KB	No result	No result

Table 9-1: Memory usage with 1000 and 4500 idle WebSockets connections.

I was not able to get any results with 4500 clients using Lightstreamer. At around 2500 clients, the browsers used so much memory that my computer crashed.

10 Analysis

In this chapter, I discuss the results of the tests. I will try to provide plausible explanations to the anomalies in the results. The meaning of the results will also be discussed. Explanations are the result of my own experiences and opinions.

10.1 Message frequency

This section discusses the results from sections 9.1, 9.2 and 9.3. Most aspects applies to all, so I found it natural to keep the discussion in one section. All tables show the highest and lowest recorded data from their respective results. These numbers are based on the raw data and not the average values from the graphs. However, the average values displayed in each table, represent the total average value. These values are derived from the graphs.

Looking at any graph about message frequency, one can see some minor drops. These are most likely the result of either concurrency issues or timing. Thread proofing any application is not an easy task, even for experienced developers. It is likely that my code has one or more weak spots in this matter. This may have caused some messages to be skipped by the logging mechanism, resulting in a minor drop in the graph.

Time issues can have surfaced even though I used a separate start time for clients and the server (as described in section 8.1). The clients start time was set before the call to the "initTest" procedure. With SignalR, I observed that this took some time to respond, which explains the one second overtime each test with this framework had. The other frameworks also had some occurrences of this, resulting in an average result with one second overtime.

Project part 2: Load testing

The table below (Table 10-1) shows data for messages sent by the clients. Most cases show an average number that is smaller than the expected 120. All these are the result of tests that went into overtime.

Framework		Polling	Long polling	Streaming	Server-Sent Events	WebSockets	WS-polling
SignalR	High:		120		120	120	
	Low:	-	2	-	18	2	-
	Average:		113		112	112	
Socket.IO	High:	120	120			120	
	Low:	2	4	-	-	120	-
	Average:	72	113			120	
SockJS	High:		120	120		120	
	Low:	-	120	4	-	4	-
	Average:		120	107		114	
Play	High:					120	
	Low:	-	-	-	-	120	-
	Average:					120	
Lightstreamer	High:	120		120		120	120
	Low:	2	-	2	-	44	1
	Average:	111		88		114	109

Table 10-1: Messages sent by clients.

While Socket.IO had 120 as highest number for polling, its average is a lot lower. As you see from Figure 9-2, it was far beneath the expected frequency throughout the test. A probable explanation to this behavior is a combination of two things. First, that the server took a long time to respond to a poll request. The captured network traffic indicate that several messages often was bundled into a single response. This operation requires time on Node's singe thread, and may have blocked other incoming requests for a short period of time. As a result, the clients may have reached six outgoing requests which is Firefox's maximum [133]. Together, these phenomena result in a queue of request forming at both the clients and the server. Then, ten seconds of overtime isn't too improbable.

The client's single threaded nature, can cause some issues. If the server has a peak in messages sent, which some results show, the client suddenly receives a lot more work. This can push subsequent "send"-calls far down the call stack, resulting in a frequency drop.

Explaining what happens towards the end of each run with Lightstreamer is rather hard. All runs show unstable frequencies. The tests I ran with all frameworks to see the resource usage with many idle clients using WebSockets may have revealed a possible reason. With Lightstreamer, I was not able to connect more than about 2500 clients before the browsers used all my computer's memory. None of the other frameworks displayed this behavior. This may indicate that there is a memory leak in the Lightstreamer client library. If this is the case, it can impact the clients abilities. But, as I have no clear indications towards this, it remains pure speculation.

Another possible explanation is the usage of an ExecutorService [94] with a cached thread pool to handle concurrency. This is the only difference between the concurrency aspects of Play and Lightstreamer. As you can see of the data, Play had no drops in frequency at all. I have found indications that a cached thread pool used in conjunction with synchronized code, degrades performance [135]. This is exactly how I registered data in the Lightstreamer application (Example 10-1). If this is the case, more than just the frequency data may have been affected–latency may have been affected as well.

```java
Runnable task = new Runnable() {
        @Override
        public void run() {
                try {
                        //Triggers synchronized code:
                        localListener.broadcast(cid, message);
                } catch (IOException e) {
                        e.printStackTrace();
                }
        }
};

executor.execute(task);
```

Example 10-1: Code executed by an ExecutorService that triggers synchronized code.

Project part 2: Load testing

The next table (Table 10-2) shows data for messages received by the server. There are some single runs with intervals with more than 120 messages received for some frameworks and transports. SignalR had this behavior for both Server-Sent Events and WebSockets. Both cases had this as the number for the interval from one to two seconds. Both followed a first second with close to 110 received messages. In section 9.4, I mentioned that SignalR could have benefitted from a warm up of the server. This behavior seems to support this.

Framework		Polling	Long polling	Streaming	Server-Sent Events	WebSockets	WS-polling
SignalR	High:		120		127	127	
	Low:	-	4	-	2	2	-
	Average:		113		112	112	
Socket.IO	High:	110	120			120	
	Low:	1	2	-	-	120	-
	Average:	72	113			120	
SockJS	High:		120	133		120	
	Low:	-	120	4	-	27	-
	Average:		120	107		113	
Play	High:					120	
	Low:	-	-	-	-	120	-
	Average:					120	
Lightstreamer	High:	269		130		120	122
	Low:	0	-	4	-	2	2
	Average:	111		88		117	100

Table 10-2: Messages received by server.

With HTTP-streaming, we see some high peaks occurring more than once. Looking at the network captures for these cases, it appears that some responses to incoming POSTs were sent rather late. It is not possible to tell for sure, since the capture says nothing about what browser a request and response belonged to. Nor does it say what specific request a response belongs to. If some responses were slow, it may have caused the clients to reach the connection limit.

Project part 2: Load testing

The final table regarding message frequency (Table 10-3), shows messages sent from the server. This data is proportional to messages received, and indicate that each server managed to send messages out at the same pace as it received them. The most obvious anomaly is also seen in the previous table.

Framework		Polling	Long polling	Streaming	Server-Sent Events	WebSockets	WS-polling
SignalR	High: Low: Average:	-	7320 60 6746	-	7620 120 6741	7620 360 6777	-
Socket.IO	High: Low: Average:	6540 120 4350	7200 120 6785	-	-	7200 7200 7200	-
SockJS	High: Low: Average:	-	7200 7200 7200	7560 240 6411	-	7200 1620 6823	-
Play	High: Low: Average:	-	-	-	-	7200 7200 7200	-
Lightstreamer	High: Low: Average:	15060 0 111	-	7920 120 5262	-	7200 120 7013	7320 120 6001

Table 10-3: Messages sent from server.

Lightstreamer had some occurrences of no messages sent or received in a given interval. In the same interval, there is no corresponding drop in the messages sent from clients. Something happened on the server that caused it to "skip" a whole second. Immediately after the dormant interval, it sends these "skipped" messages, resulting in the high peaks you see in the table (as much as 15060 sent messages with polling).

Network captures from these runs show that it is not because of a registration error. In the intervals in question, there actually are no messages going out from the server. At the same time, only a few POST requests are sent as well. Since there is no drop in send frequency, it seems that the send timestamp has been set correctly. The only explanation then is that something happened with the "send" routine of Lightstreamer's client library. This may have resulted in a delay for most of the POST requests.

10.2 Average latency

This section discusses the latency results. The results presented each transport individually. Here I will discuss the performance of each framework, but the main focus will be on comparing the transports.

The table and graph presented in this section used the average values from the fifth second to the 14th–a ten second span[43]. There are two main reasons for this: Some frameworks had anomalies in the first couple of seconds, indicating that they could have benefitted from a warm up period. Others had anomalies towards the end with a lot of overtime. The most representative results are in the interval that the table is based on.

10.2.1 Frameworks

The relationship between the frameworks is consistent for almost all the results (Table 10-4). Socket.IO and polling is the only combination that deviates. Otherwise, each framework's "rank" using WebSockets remains the same across all transports:

- 1st: Socket.IO.
- 2nd: Lightstreamer.
- 3rd: SignalR.
- 4th: SockJS.
- 5th: Play.

Framework		Polling	Long polling	Streaming	Server-Sent Events	WebSockets	WS-polling
SignalR	High:		155,4		27,1	23,6	
	Low:	-	113,7	-	14,1	14,2	-
	Average:		134,8		18,1	18,4	
Socket.IO	High:	666,8	58,1			7,4	
	Low:	553,5	48,8	-	-	5,3	-
	Average:	605,6	52,6			6,0	
SockJS	High:		223,3	172,9		63,1	
	Low:	-	215,3	86,5	-	59,2	-
	Average:		198,6	120,0		61,2	
Play	High:					85,4	
	Low:	-	-	-	-	69,9	-
	Average:					75,9	
Lightstreamer	High:	247,9		49,5		29,0	79,5
	Low:	98,1	-	22,8	-	12,9	44,4
	Average:	161,7		33,1		17,8	61,8

Table 10-4: Average latency in milliseconds.

[43] This represents the interval from 5 seconds up t0, but not including 15 seconds.

Even though Socket.IO ran 10 seconds too long using polling, it is a little strange that the latency is so much more than long-polling. Lightstreamer's latency increases by a factor of 4,89 from streaming to polling[44]. Moving from long-polling to polling shows a factor of 11,51[45] for Socket.IO. Inspecting the network traffic reveals a probable cause. Both frameworks bundle several messages into single responses. But the captures for Lightstreamer has almost double the amount of packages. This indicates that Socket.IO bundles more messages into single responses than what Lightstreamer does. Doing this costs time, resulting in the higher latency.

Socket.IO in general performs better than Lightstreamer and SignalR in the tests. With a higher load this would probably not be the case. A single threaded server will most likely not be able to handle the same amount of load as one with many threads. On the other hand, one can run several Node instances for the same resource cost as a single IIS or Lightstreamer server. This case is outside the scope of this thesis.

Both Play and SockJS are more manual approaches. SockJS handles many real time features, but broadcasting is not something it supports out of the box. These frameworks relied on a loop to send messages to all clients, a solution that is obviously not optimal. As a result, they are far behind the others. An observation that is very clear if you compare SockJS to Socket.IO. Both run on Node, but Socket.IO is a lot faster. Even Socket.IO's long-polling beats SockJS's WebSockets.

10.2.2 Transports effect on latency

The graph in Figure 10-1 shows the average values from Table 10-4. To make it more readable, I've cut down the y-axis. The bar representing polling with Socket.IO is far above the axis in this graph. The original graph can be viewed in Appendix B.

Across all frameworks, WebSockets is the transport that performs best. The only exception is Server-Sent Events that perform just as well[46]. HTTP-streaming isn't far behind, but here it's a clear increase in latency. Long-polling and polling are, without doubt, a lot slower than any streaming technique–WebSockets or not.

---

[44] 161,7 / 33,1 = 4,89.
[45] 605,6 / 52,6 = 11,51
[46] Server Sent Events actually beats WebSockets by 0,3 milliseconds. As this is close to 0, I deem them just as good.
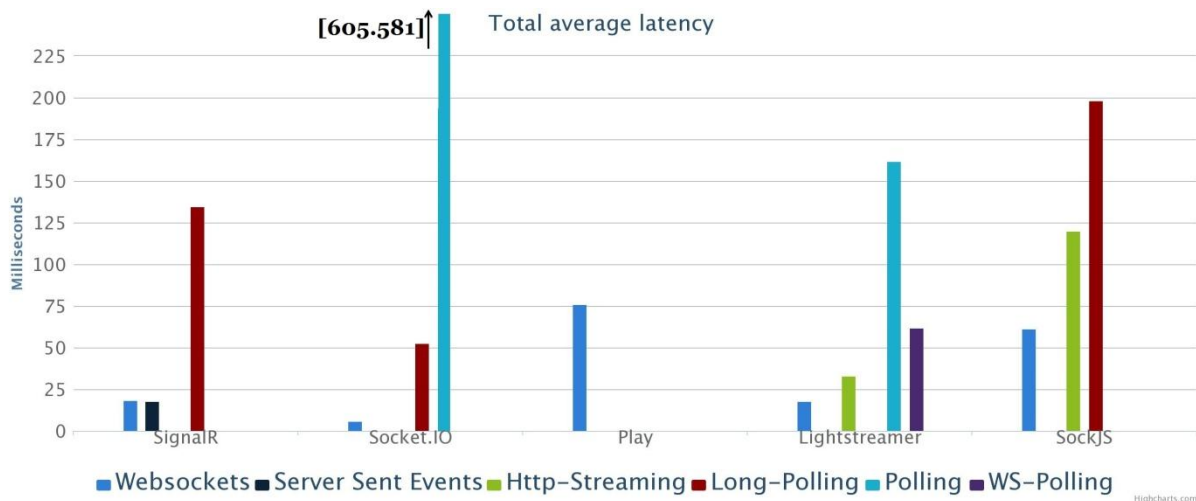
Figure 10-1: Average latency as bar chart.

The low latency of HTTP-streaming surprised me a little. It has an average latency of about twice as much as WebSockets. I expected this to be more. A large part of this difference is likely because it takes time to send a message from a client to the server. This uses a normal HTTP-request, which relies on setting up and tearing down a whole new connection each time. It is clear to me that HTTP-streaming is a reliable technique to build pure push applications.

Considering that Server-Sent Events also relies on HTTP-requests to get messages from a client, its performance is impressive. My tests don't have a high load, though. Higher loads may benefit using WebSockets instead of Server Sent Events. A blog post by William P. Riley-Land suggests just that [136]. This test used two different libraries for the different transports: connect-sse [137] for Server-Sent Events and Socket.IO for WebSockets.

Riley-Land claims that the results are *"in the same order of magnitude"* [136]. His results show that WebSockets are 31%[47] faster. I disagree a little with his conclusion as I consider 31% a clear difference. However, with the substantial load in his tests, I can see that the difference in latency per message is quite small. Another blog post [138] showed complete opposite results–showing that Server-Sent Events were better. This used a proxy with beta stage support for WebSockets, so I don't count these results as reliable.

---

[47] WebSockets averaged 374,64 milliseconds. Server Sent Events averaged 490,44 milliseconds. 374,64 + 31% = 490,44.

Server-Sent Events and WebSockets are both HTML5 APIs. A question that came to my mind is: Do we need both?. Server-Sent Events is simpler to implement than WebSockets. It has a more powerful API and server side it uses normal HTTP. With that in mind, I also support this opinion. But with a framework such as SignalR, this argument is invalid, since it handles the transport of messages for you. Still, for pure push applications, I can see the benefit of using Server-Sent Events.

The use of WebSockets to do polling is not meant for a push dominated application. I don't see why it is part of the Lightstreamer stack, since the connections are kept open just as with streaming. The only difference is that the client has to send a poll message over the WebSocket connection to get data. The result is that the performance drops and streaming over HTTP becomes a better alternative.

A browser that supports WS-polling, obviously supports streaming over WebSockets. For real time purposes, this is preferable anyways, which renders WebSockets polling little useful. To me, it was nice to have, as it helps highlight differences between HTTP and WebSockets. Polling over WebSockets is almost three times faster than over HTTP (based on the Lightstreamer results).

10.3 Machine resources

This section will focus on the resource usage of the different transport mechanisms in the context of the frameworks. Section 8.11 describes the differences between the various platforms. Implications of this are that it is hard to compare for instance SignalR to Socket.IO or even Lightstreamer for that matter. Still, I will provide a short discussion regarding this.

Table 10-5 gives an overview of resource usage data collected from the graphs in Figure 9-13 and Figure 9-14. There are some interesting aspects in these results. SockJS uses more processor than Socket.IO with WebSockets, but less with long-polling. Except for this, the general trend is the same as for latency: the "rank" with WebSockets remains for the other transports.

SockJS uses a library for its WebSockets support: Faye [139], whereas the rest is built from scratch. But since long-polling uses less than HTTP-streaming as well, I do not think this is the reason. It may be part of it. How Node handles HTTP-streaming may also be the cause. But since Socket.IO didn't support HTTP-streaming, there is no way to know for sure.

Framework		Polling	Long polling	Streaming	Server-Sent Events	WebSockets	WS-polling
SignalR	Processor: Memory:	-	58,7% 171 KB	-	37,6% 165 KB	37,9% 144 KB	-
Socket.IO	Processor: Memory:	8,2% 32 KB	14,8% 32 KB	-	-	14,3% 35 KB	-
SockJS	Processor: Memory:	-	10,7 % 32 KB	21,9% 31 KB	-	17,1% 32 KB	-
Play	Processor: Memory:	-	-	-	-	32,1% 137 KB	-
Lightstreamer	Processor: Memory:	70,2% 888 KB	-	26,2% 107 KB	-	21,6% 110 KB	47,8% 209 KB

Table 10-5: Resource usage of all frameworks and transports.

Another thing that stands out is that Play uses more resources than Lightstreamer. As Play is the most bare implementation, I did not expect this. Obviously, the Lightstreamer server has a more lightweight implementation than the server Play uses.

It is a trend in the data that the use of machine resources increases as the transport moves farther from WebSockets[48]. Except from polling with Lightstreamer it's nothing dramatic. But there are some exceptions. Some transports even use a little less of either processor or memory than WebSockets. This difference, is so small that it can be counted as equal. This means that for example Server-Sent Events use just as much processor as WebSockets.

WS-polling use twice the amount of resources as streaming with WebSockets with Lightstreamer, which is a little strange. That it uses more processing power isn't too unlikely, but the extra memory is. It uses the same amount of open connections as streaming with WebSockets. Handling polling may require some sort of mechanism that streaming doesn't need. This can explain the extra memory usage. But it does not explain why polling over HTTP takes up more than four times as much as over WebSockets.

Despite the presence of a garbage collector, memory leaks can occur in Java programs [140]. With an immature framework, I could have believed this to be the cause. But with a 14 year old framework, it is hard to believe. No other combination of transport and framework shows the same increase in resource usage as with Lightstreamer. In my opinion, this makes a memory leak plausible.

---

[48] As you move through Server-Sent Events, HTTP-streaming, long-polling and polling you get farther from WebSockets.

Unfortunately, the polling results of Socket.IO cannot be deemed accurate. The server used less resources, but as the test took ten seconds longer, the results cannot be compared to Lightstreamer, which is the only other framework with support for polling in Firefox.

10.4 Network performance of frameworks

The results show only small differences between most of the frameworks. SignalR, on the other hand, stands out both in the calculated and the captured results. Looking into the capture files, one can see that this is a manner of how it handles messages. There are some excess data like cursors that the other frameworks don't have in the same way. SignalR uses a cursor to help with reconnection. This may give SignalR an edge over the other frameworks in this matter. Investigating this is outside the scope of this thesis, which is why I haven't looked into it.

10.5 Transports effect on network traffic

In section 9.7 I showed figures displaying the captured network traffic and the calculated. The table below (Table 10-6) compares the data from the different results.

		Polling	Long polling	Streaming	Server-Sent Events	WebSockets	WS-polling
SignalR	Calculated: Captured:	- 	220M 49M	95M -	94M 45M	65M 45M	-
Socket.IO	Calculated: Captured:	105M 31M	104M 30M	- 	- 	27M 30M	-
SockJS	Calculated: Captured:	- 	125M 27M	29M 27M	70M -	27M 31M	-
Play	Calculated: Captured:	- 	- 	36M -	- 	28M 26M	-
Lightstreamer	Calculated: Captured:	188M 65M	- 	42M 32M	- 	29M 29M	68M 36M

Table 10-6: Both captured and calculated network traffic.

As long-polling and polling don't have an open connection throughout the test, the captures of these two transports should be correct. The case with SignalR shows that these cannot be trusted either. Still, the difference between the calculations and the captured results are substantial for most transports. WebSockets actually turned out to be closest to the theoretical throughput.

Using a short run with only one client as basis for calculating, introduces possible differences with a full run. I have already described behavior where several messages have been put into single responses (see section 10.1). This is a common way of saving network usage, and it is likely that all frameworks do this across all transports.

Project part 2: Load testing

So the calculated data does not take message compression into consideration. Nor does it consider any altered behavior in cursor messages such as SignalR sends. But the data is an accurate representation of the theoretical throughput. The only presumption is that communication remains the same. Keeping this in mind, comparing frameworks and transports on this basis is a valid method in my opinion.

Chapter 3 highlighted overhead regarding header data as a drawback of HTTP. Looking at the results may cause you to believe that this is false for HTTP-Streaming and Server-Sent Events. Let us use SignalR as an example. The capture used to calculate throughput contained 14960 bytes for WebSockets and 45879 bytes for long-polling. In other words 3,1 times as many bytes. The calculated data show 64,98 and 219,69 million bytes for WebSockets and long-polling respectively. Long-polling has a theoretical throughput of 3,4 times as many bytes. Since the relationship remains close to the same, it seems that the overhead-claim is correct. But if we look at Server-Sent Events, we see that the results are different.

29066 bytes was captured with Server-Sent Events. More than twice the amount of bytes as for WebSockets. But the calculations show 93,57 million bytes, only 1,4 times as much. This makes it seem that the overhead has decreased.

If we look at the behavior of the two cases, the capture and the full test, the reason becomes clear. The small test used for the capture sent the same amount of messages as it received back at the clients. In the full test, as you can see from Figure 10-2, this is not the case at all. Here, each message sent to the server results in a broadcast to all clients.
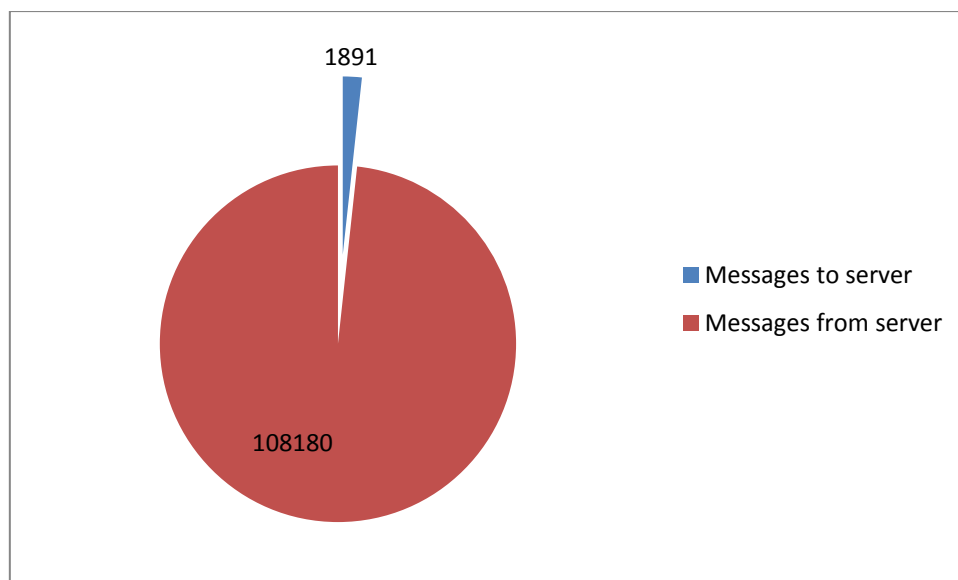
Figure 10-2: Piechart that shows relationship between messages to and from the server.

Since messages going from the server use an already open connection, there is no header data. Only the messages from the clients has this. With non-streaming techniques, receiving from the server involves a GET or POST request first. Then you get the extra overhead also for the messages going from the server.

This does not implicate that pure push HTTP-technologies match WebSockets's byte usage. Every framework "spends" more bytes sending a message over a streaming connection than over WebSockets. Most of the frameworks show that HTTP-streaming or Server-Sent Events use around 30-50% more on average. SockJS is an exception to this. Here HTTP-streaming use the same as WebSockets. As the WebSockets part of SockJS use another library, it is not possible to tell if this is representative.
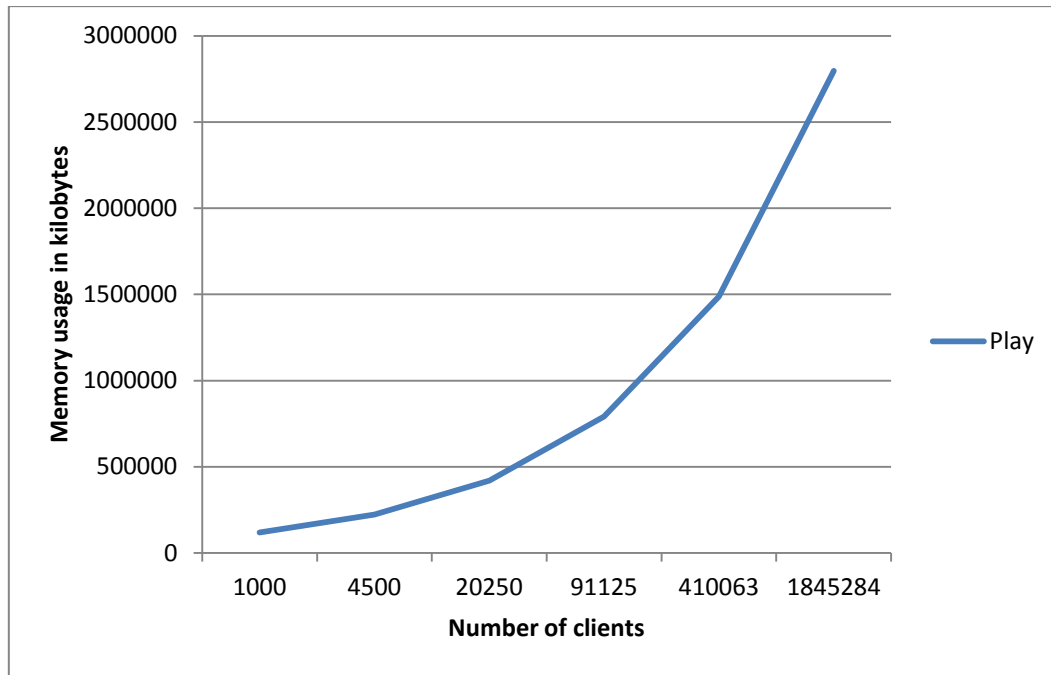
10.6 Idle clients with WebSockets

Memory usage does not increase with the same factor all the time as new clients connect. But for a worst case scenario, we can pretend it does. Recall the results from Table 9-1 in section 9.8. SignalR was the framework that showed the largest increase in memory usage. It also used the most memory to begin with. Since I am already considering a worst case scenario for memory growth, I will not use the "worst" framework as basis. Furthermore, I don't see a future where the majority of servers are Node-based. One of the Java frameworks will therefore be more representative. So I will use Play as an example, as this is close to a bare real time implementation.

The graph in Figure 10-3 shows how the memory usage will develop if it increases with a factor of 1,88 while clients increase with a factor of 4,5. When it closes on two million clients, it approaches a memory usage of 2,9 gigabytes[49]. Two million concurrent clients is a lot for most web pages. But, if a static web page has this many idle clients at once, it will not affect the server at all if it uses traditional HTTP. If it uses WebSockets to serve content to all clients, the idle connections take up 2,9 unnecessary gigabytes of memory.

---

[49] 3.000.000 kilobytes /1024² = 2,9 gigabytes.

Figure 10-3: Theoretical development of memory usage of idle WebSockets clients with Play.

This thesis has established the fact that WebSockets is faster than request/response based HTTP. But seeing the extra memory usage, the tradeoff is not large enough. Furthermore, my tests focused on real time use cases. For a web page without real time content, the differences in latency would be minimal.

Another benefit of WebSockets we have seen is that it has less overhead. Using the protocol as basis for a new standard for all client/server communication, much of this benefit disappears. WebSockets does not have headers in the same sense as HTTP. To implement a similar request/response pattern as HTTP with WebSockets, this would have to be added to the protocol.

These are the main reasons why WebSockets is a separate protocol designed for real time only. The background gave a brief overview of the upcoming HTTP/2.0 specification (see section 2.3). While this specification proposes changes to HTTP, including push behavior [21], it does not mention WebSockets.

In my opinion, all web pages should use bundling and minification of CSS and JavaScript. This means that all the code is compressed into two single files: one for CSS and one for JavaScript. As a result, a browser can get a whole web page using a minimum of three requests. Any images, videos, etc. will increase this number. Doing this allows for better utilization of HTTP. This makes the benefits of WebSockets decrease even more for static web.

While WebSockets may never become the basis for a new generation of HTTP, the results from the idle connections experiment has a side effect. Idle WebSockets connection take up resources. Some real time applications may not need to send updates to the clients all the time. With a set, rather large, interval that developers know, polling might actually be a better choice. No resources will be wasted serving idle connections, and users get their data when they need it.

At such low loads, the extra network traffic becomes insignificant. An application with an interval so large that polling becomes better than WebSockets is not a common case. To keep the benefit, messages going from the client to the server should be in the same order of magnitude. Introducing broadcasting quickly makes WebSockets a better choice with respect to network traffic.

Conclusion

11 Frameworks

This chapter covers the problem statements regarding the different frameworks. The problem statements are reproduced in the paragraphs below. Before drawing up the final conclusion for each, I will summarize experiences made during the project.

I: How are the frameworks that are featured in this thesis with respect to usability?

Socket.IO is built according to the mantra of Node [62]. It is a lightweight library for real time. The communication interface uses events, which are easy to relate to and work with in JavaScript. Writing automated tests for event driven applications can be tricky. This is no issue with Socket.IO as it provides a client library for Node that you can use for integration tests. There are also ways of writing unit tests.

The RPC model of the .NET library SignalR is clever and intriguing. Calling a function from the server and vice versa, is even simpler to relate to than events. Some frameworks that has "magic" going on behind the scenes like SignalR, can get confusing. With SignalR, this is not the case, especially because of the thorough documentation. Plenty of tutorials makes getting started with SignalR a breeze. And if you are an experienced .NET developer, you should have no trouble using it.

Lightstreamer is complicated compared to the simple models of SignalR and Socket.IO. For the work with my application, it was actually no easier to use than the close to bare metal Play! Framework. But as Lightstreamer has support for more elements, such as more fallback transports and reconnection, it is a little simpler than Play in the long run. With Play, all fallbacks have to be implemented manually. Lightstreamer is a very mature product, but I feel it needs to be changed in order to compete with lightweight libraries for real time.

Meteor is an interesting piece of technology. It will be exciting to follow its development in the future. Right now, it is far from complete. This makes it not too enjoyable to work with. The lack of a testing framework is a cause of concern, which makes it hard to predict the maintainability of Meteor apps when the framework reaches a stable varsion.

Table 11-1 shows a score representation of each usability aspect for all frameworks. I have set the score on a scale from one to ten, based on the results described in chapter 7. One small note is that while Lightstreamer is the oldest framework by far, it has a maturity score of only 8. This reflects that I think it has become too mature.

Conclusion

Framework	Documentation	Simplicity	Maintainability	Browser support	Maturity
Socket.IO	7	10	9	10	8
Lightstreamer	8	6	9	10	8
Play	9	5	8	10	7
SignalR	9	10	9	10	9
Meteor	6	7	5	10	3

Table 11-1: Each framework's overall score for each usability criteria.

SignalR and Socket.IO stand out as the most enjoyable frameworks to work with. They both provide a simple, yet powerful programming model. Fallbacks to older transports, as well as serialization of data, happens behind the scenes no matter which you use. As I find SignalR's documentation to be a little better than Socket.IO's, SignalR is the most usable framework in my opinion. It is also a little more mature.

II: Which of the frameworks have the best performance in terms of message frequency, latency, network usage and server resource consumption?

The performance tests showed Socket.IO as winner in the latency category. Lightstreamer and SignalR follow a little behind with a large gap down to SockJS and Play. This was the tendency of the results regarding use of machine resources as well. Both Node frameworks used a lot less than any of the others. Node is single threaded, which implicates that Socket.IO doesn't scale as well as for example Lightstreamer or SignalR. But since it is such a light weight platform, one can have several instances running for the same cost as one IIS or Lightstreamer server.

When it came to network performance, Socket.IO, once again, was on the top. However, it does not support HTTP-streaming as fallback. As a result, all frameworks first fallback[50] is better than Socket.IO's.

SignalR is behind all the others when it comes to network usage, which is due to how it handles messages. Other data than just the message payload, like cursors, is sent with each message.

Table 11-2 shows a similar score representation as Table 11-1 for the different performance aspects. I have left message frequency out as there were some anomalies in this category. These anomalies were probably because of my test setup or implementation. As all the frameworks have different transports, WebSockets performance was the main basis of the scores.

---

[50] The "first" fallback is the first transport a framework will try to fall back to if WebSockets is not supported.

Conclusion

Framework	Latency	Network	Memory	Processor
Socket.IO	10	10	10	10
Lightstreamer	9	9	8	9
Play	3	10	8	8
SignalR	9	7	8	8
SockJS	5	9	10	10

Table 11-2: Each framework's score for the different performance aspects.

Based on the tests performed in this thesis, Socket.IO has the best performance regardless of platform. Lightstreamer is not far behind and it is the best Java-based framework. It is also better than SignalR, the only C# framework featured in this thesis. SignalR ran on IIS8 for all the tests. Using OWIN to host the server might increase performance with respect to memory consumption and processor usage.

III: Are there any real time web applications that may benefit from not using the aid a framework provides?

Play gave me insight into how it is to implement real time almost without support. This was a far more complicated process than with most of the other frameworks. Lightstreamer was actually not far behind the complexity of the Play application. But when you look at the whole picture, Lightstreamer offers a lot more than Play. Play has no client handling, no serialization, no reconnect mechanism and no fallbacks out of the box. My application only offered two fallbacks. Introducing more, would introduce more complexity.

However, there is another aspect to consider when it comes to building real time applications. File sizes may be important to certain types of scenarios. For instance if you are developing for mobile devices or if the application will be deployed in a low bandwidth environment. As the client side code with Play is without any library dependencies, it is a lot smaller in size than with any of the other frameworks.

This leads me to the conclusion that you want to use a framework if the size of client files offered by it is acceptable. Even if fallbacks are not required, it is harder to build real time functionality manually. A framework also provides more flexibility. If fallbacks become a requirement after all, it is already there.

12 WebSockets versus HTTP

In this chapter I will focus on the problem statements that pin WebSockets against HTTP. As with the previous chapter, these are reproduced in the paragraphs below.

IV: Does WebSockets outperform the old, established HTTP methods for real time in terms of network usage, message latency and use of machine resources?

WebSockets outperform most of the HTTP methods when it comes to latency. A surprising exception in my tests was Server-Sent Events. But, as I discussed in section 10.2.2, it is likely that WebSockets is better with higher loads. HTTP-streaming isn't that much slower, but it is noticeable.

My tests had a strong focus on server to client communication. The other way around, HTTP relies on normal requests. Long-polling and polling are a lot slower than WebSockets. Therefore, I have reason to believe that WebSockets is far better for applications with a higher percentage of client to server communication.

This also applies to network traffic. HTTP-streaming and Server-Sent Events almost keep up with WebSockets in push focused applications. The more client to server communication you get, the bigger the benefit of WebSockets.

For real time applications with high message frequency, WebSockets require less memory and processor than HTTP. As the frequency decreases, the more memory the server wastes serving idle connections. With a large enough interval between updates, that the developers know up front, polling can be a good choice. There are not too many use cases that are like this, however.

Based on the observations I have made, WebSockets is better than HTTP in every aspect for real time applications. But for pure push applications, I will not say that WebSockets outperforms HTTP, even though it is better. With respect to client to server communication, there is no doubt that WebSockets is faster and use less resources. Some rare cases with a large push interval and little client to server communication, can cause polling to be a better choice than WebSockets.

V: Can WebSockets be the foundation for the next generation of HTTP?

Idle WebSockets connections take up memory. With the request/response pattern of HTTP, idle connections don't exist. Using WebSockets as a foundation for a HTTP protocol would mean trading a significant increase in memory usage for faster connections. As header data would have to be added to WebSockets, savings on network performance will be minimal.

HTTP/2.0 incorporates an idea from real time, with the ability to push data to clients. But the protocol remains unidirectional. If developers take care to bundle and minify CSS and JavaScript, the loading of web pages will become faster with HTTP/2.0.

Conclusion

Meteor uses real time for every aspect of client/server communication, but it serves files via normal HTTP. Still, it seems that the developers believe in WebSockets for all communication in their applications. With real time so central, Meteor is most suitable for applications with many real time components in my opinion.

HTTP/2.0 surfaced during the initial research for this thesis. At the time, I saw benefits of using WebSockets instead of HTTP for some aspects. But with the current draft of HTTP/2.0, I cannot see a future where WebSockets become an all purpose protocol. It was built as a protocol for real time, and that is what it should remain in my opinion.

13 Further work

This thesis has a lot of potential for further work. In this chapter, I describe possible projects that can build upon the work I have done.

13.1 Scalability

My work focused on usability as well as performance. In retrospect, it would be interesting to focus on performance alone. The tests I performed show clear trends between frameworks and transports. But important aspects, such as scaling, are not covered.

To be able to handle this, one have to change the approach. Using real browsers is not an alternative. The preferable client is a headless browser with full support for WebSockets. In a not too distant future, both Slimer and Phantom should be applicable. If not, the use of console applications is almost just as good. Both will allow for more clients on a single machine. Even more if you use more machines.

If I were to do this project, I would have changed the way I collected some of the data. This thesis used message frequency to see if and when messages were lost. For a larger test, one should only measure the "if"-part. In other words, the server would just count the number of messages, not divide them into time intervals.

13.2 Separate message directions

The push nature of my tests had a large influence on the results. It seems that HTTP push technologies are not far behind WebSockets in performance. The other way, the results from long-polling and polling indicate that WebSockets is a lot better. But I did not perform any tests to find out for sure.

An interesting case would be to run tests with many more clients than I had, that focus on just one message direction at a time. One test for server to client and one for client to server. The first will compare WebSockets to HTTP-streaming and Server-Sent Events. From client to server you need to compare WebSockets to either POST or GET requests.

A project of this nature should not focus on different frameworks. To make it as even as possible, one should keep to one platform.

13.3 Cluster of Node servers

I've paid a lot of attention to the lightweight nature of Node. In my tests, this was a huge benefit for both Socket.IO and SockJS in terms of resource usage. However, with higher loads, the single thread of Node should be overwhelmed. Node uses little machine resources compared to both Java and C# counterparts. An interesting project would be to perform load tests where one compares a cluster of Node servers to for instance Lightstreamer and SignalR.

The ideal setup would have the same amount of Node servers as the other frameworks have available threads. All Node servers should run on the same machine. Further testing can involve the use of different load balancers for the Node servers.

SignalR does not depend on IIS. It can use OWIN for self hosting [141]. This should make the server a lot more nimble. Testing this versus both Lightstreamer and a Node cluster could make this project even more interesting.

Depending on the time at hand, the work you can do with this is enormous. Further expansion can use a more advanced setup. You can introduce several servers for Lightstreamer and SignalR, running on several machines. The same machines can host one Node cluster each.

This would require many clients to be able to get viable data. Maybe as many as 100000 or more. Hence, you will need access to a great number of machines. The way you collect data will also have to be changed to fit the setup. A simple solution is to have the servers send a set number of messages as fast as possible, and then measure the time. You can also turn this around and have the clients send messages to the servers.

13.4 Atmosphere

Over the last year and a half, Atmosphere has gained popularity. Lightstreamer was my Java alternative for real time. This framework has a different approach than Atmosphere. Atmosphere builds on the same concepts as Socket.IO and SignalR.

Using those three frameworks as basis for a project, one could get a true showdown of platforms; Java, Node and ASP.NET/C#. One can use several approaches, but I think there should be a strong focus on performance. During my work, I learned a lot about the usability of the frameworks during the performance tests. A project with a load test of these three frameworks would require development with each. Experiences from this work can then be shared in the same manner as in part one of my project (see from page 33).

13.5 HTTP/2.0

"Can HTTP/2.0 improve existing HTTP methods for real time?" This would be an interesting problem statement for a project based on this thesis. Doing the same, small scale tests, one can compare my data to data recorded for HTTP/2.0. To get a better picture, one can perform more large scale tests focusing on push.

A technique like long-polling should be able to benefit a lot from the push mechanism in HTTP/2.0. Maybe it can be used almost like Server-Sent Events? If that is the case, one should see how it performs. A result showing that long-polling over HTTP/2.0 perform just as well as Server-Sent Events, would be intriguing.

Sources

Sources

[1]   P. Leggetter, "Real-Time Web Technologies Guide." [Online]. Available:
        http://www.leggetter.co.uk/real-time-web-technologies-guide. [Accessed: 25-
        Mar-2014].

[2]   N. Podbielski, "WebSocket libraries comparison - CodeProject." [Online].
        Available: http://www.codeproject.com/Articles/733297/WebSocket-libraries-
        comparison. [Accessed: 25-Mar-2014].

[3]   "Real-Time Analytics with WebSockets: SignalR vs. Node.JS with Socket.IO –
        Round 1." [Online]. Available: http://sim4all.com/blogging/?p=454.
        [Accessed: 25-Mar-2014].

[4]   "SockJS, multiple channels, and why I dumped socket.io | Matt's Hacking Blog
        on WordPress.com." [Online]. Available:
        http://baudehlo.com/2013/05/07/sockjs-multiple-channels-and-why-i-
        dumped-socket-io/. [Accessed: 25-Mar-2014].

[5]   "SockJS vs Socket.IO - Benchmarked - PythonAnywhere News." [Online].
        Available: http://blog.pythonanywhere.com/27/. [Accessed: 25-Mar-2014].

[6]   XSockets, "XSockets vs SignalR | xsockets.net." [Online]. Available:
        http://xsockets.net/xsockets-vs-signalr. [Accessed: 25-Mar-2014].

[7]   Lightstreamer, "The Lightstreamer Blog: Benchmarking Socket.IO vs.
        Lightstreamer with Node.js." [Online]. Available:
        http://blog.lightstreamer.com/2013/05/benchmarking-socketio-vs-
        lightstreamer.html. [Accessed: 25-Mar-2014].

[8]   E. Bozdag, A. Mesbah, and A. van Deursen, "A Comparison of Push and Pull
        Techniques for AJAX," *2007 9th IEEE International Workshop on Web Site
        Evolution*, pp. 15–22, Oct. 2007.

[9]   M. Jõhvik, "Push-based versus pull-based data transfer in AJAX applications,"
        University of Tartu, 2011.

[10]   D. G. Puranik, D. C. Feiock, and J. H. Hill, "Real-Time Monitoring using AJAX
        and WebSockets," *ECBS '13 Proceedings of the 20th Annual IEEE
        International Conference and Workshops on the Engineering of Computer
        Based Systems*, pp. 110–118, 2013.

[11]   X. Wang, "Library vs. Framework?" [Online]. Available:
        http://www.programcreek.com/2011/09/what-is-the-difference-between-a-
        java-library-and-a-framework/. [Accessed: 20-Mar-2014].

[12]   M. Simpson, "Unit? Integration? Functional? Wha? | Mark's Devblog."
        [Online]. Available: http://defragdev.com/blog/?p=611. [Accessed: 20-Mar-
        2014].

[13]   K. Johannessen, "Code base - GitHub." [Online]. Available:
        https://github.com/kjohann/MasterThesis.

Sources

[14]   I. Peter, "History of the world wide web," 2004. [Online]. Available:
        http://www.nethistory.info/History of the Internet/web.html. [Accessed: 24-
        Jan-2013].

[15]   R. Fielding, H. Frystyk, and T. Berners-Lee, "Hypertext Transfer Protocol --
        HTTP/1.0," 1996. [Online]. Available:
        http://www.w3.org/Protocols/HTTP/1.0/draft-ietf-http-spec.html. [Accessed:
        23-Jan-2013].

[16]   H. W. Lie and B. Bos, "The CSS saga," in in *Cascading Style Sheets, designing
        for the Web*, 2nd ed., 1999.

[17]   K. Balachander, C. K. Jeffrey, and M. M. David, "Key Differences between
        HTTP/1.0 and HTTP/1.1." [Online]. Available: http://www8.org/w8-
        papers/5c-protocols/key/key.html. [Accessed: 22-Jan-2013].

[18]   R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T.
        Berners-Lee, "Hypertext Transfer Protocol -- HTTP/1.1," 1999. [Online].
        Available: http://www.w3.org/Protocols/rfc2616/rfc2616.html. [Accessed: 23-
        Jan-2013].

[19]   H. Nielsen and J. Gettys, "Network performance effects of HTTP/1.1, CSS1, and
        PNG," *ACM SIGCOMM …*, pp. 155–166, 1997.

[20]   M. Belshe and R. Peon, "SPDY Protocol - Draft 3.1 - The Chromium Projects."
        [Online]. Available: http://www.chromium.org/spdy/spdy-protocol/spdy-
        protocol-draft3-1. [Accessed: 20-Mar-2014].

[21]   M. Belshe, A. Melnikov, M. Thomson, and R. Peon, "Hypertext Transfer
        Protocol version 2.0." [Online]. Available: http://tools.ietf.org/html/draft-ietf-
        httpbis-http2-00. [Accessed: 13-Mar-2014].

[22]   "Hypertext Transfer Protocol Bis (httpbis) - Charter." [Online]. Available:
        https://datatracker.ietf.org/wg/httpbis/charter/. [Accessed: 13-Mar-2014].

[23]   B. Chacos, "Next-gen HTTP 2.0 protocol will require HTTPS encryption (most
        of the time) | PCWorld." [Online]. Available:
        http://www.pcworld.com/article/2061189/next-gen-http-2-0-protocol-will-
        require-https-encryption-most-of-the-time-.html. [Accessed: 13-Mar-2014].

[24]   P. Lubbers, B. Albers, and F. Salim, *Pro HTML5 Programming*, 2nd ed.
        Springer, 2011, pp. 159–191.

[25]   D. Crane and P. McCarthy, *Comet and Reverse Ajax:The Next-Generation Ajax
        2.0*. Heidelberg: Springer-Verlag, 2008, pp. 25, 40–41.

[26]   D. Schiemann, "Comet Daily > Blog Archive > The forever-frame technique,"
        2007. [Online]. Available: http://cometdaily.com/2007/11/05/the-forever-
        frame-technique/. [Accessed: 29-Jan-2013].

Sources

[27]   A. Russel, "Comet: Low Latency Data for the Browser – Infrequently Noted."
       [Online]. Available: http://infrequently.org/2006/03/comet-low-latency-data-
       for-the-browser/. [Accessed: 29-Jan-2013].

[28]   M. Nesbitt, A. Russel, G. Wilkins, and D. Davis, "The Bayeux Specification,"
       2007. [Online]. Available: http://svn.cometd.com/trunk/bayeux/bayeux.html.
       [Accessed: 29-Jan-2013].

[29]   G. Roth, "HTML5 Server-Push Technologies, Part 1 | Java.net," 2010. [Online].
       Available: http://today.java.net/article/2010/03/31/html5-server-push-
       technologies-part-1. [Accessed: 02-Feb-2013].

[30]   E. Bidelman, "Stream Updates with Server-Sent Events - HTML5 Rocks," 2011.
       [Online]. Available:
       http://www.html5rocks.com/en/tutorials/eventsource/basics/. [Accessed: 24-
       Jan-2013].

[31]   I. Hickson, "Server-Sent Events," 2012. [Online]. Available:
       http://dev.w3.org/html5/eventsource/. [Accessed: 24-Jan-2013].

[32]   I. Fette and A. Melnikov, "The WebSocket Protocol." [Online]. Available:
       http://tools.ietf.org/html/rfc6455. [Accessed: 29-Jan-2013].

[33]   Walker-Morgan, "WebSockets becomes proposed standard," 2011. [Online].
       Available: http://www.h-online.com/open/news/item/WebSockets-becomes-
       proposed-standard-1394315.html. [Accessed: 29-Jan-2013].

[34]   P. Lubbers, B. Albers, and F. Salim, "WebSocket.org | About WebSocket."
       [Online]. Available: http://www.websocket.org/aboutwebsocket.html.
       [Accessed: 24-Jan-2013].

[35]   I. Hickson, "The WebSocket API," 2012. [Online]. Available:
       http://dev.w3.org/html5/websockets/. [Accessed: 24-Jan-2013].

[36]   P. Lubbers and F. Greco, "WebSocket.org | The Benefits of WebSocket."
       [Online]. Available: http://www.websocket.org/quantum.html. [Accessed: 24-
       Jan-2013].

[37]   "Internet Explorer Browser." [Online]. Available:
       http://www.w3schools.com/browsers/browsers_explorer.asp. [Accessed: 31-
       Jan-2013].

[38]   SignalR, "SignalR: The Official Microsoft ASP.NET Site." [Online]. Available:
       http://www.asp.net/signalr. [Accessed: 27-Aug-2013].

[39]   LearnBoost, "Socket.IO: the cross-browser WebSocket for realtime apps."
       [Online]. Available: http://socket.io/. [Accessed: 31-Jul-2013].

Sources

[40] P. Lubbers, "How HTML5 Web Sockets Interact With Proxy Servers," 2010. [Online]. Available: http://www.infoq.com/articles/Web-Sockets-Proxy-Servers. [Accessed: 29-Jan-2013].

[41] "PersistentConnection - SignalR GitHub Wiki." [Online]. Available: https://github.com/SignalR/SignalR/wiki/PersistentConnection. [Accessed: 27-Aug-2013].

[42] "Atmosphere - GitHub." [Online]. Available: https://github.com/Atmosphere/atmosphere. [Accessed: 24-Mar-2014].

[43] "Async-IO.org: Powering the Atmosphere Framework." [Online]. Available: http://async-io.org/tutorial.html. [Accessed: 20-Mar-2014].

[44] "Sails.js | Realtime MVC Framework for Node.js." [Online]. Available: http://sailsjs.org/#! [Accessed: 20-Mar-2014].

[45] "Sails.js | Realtime MVC Framework for Node.js." [Online]. Available: http://sailsjs.org/#!documentation. [Accessed: 20-Mar-2014].

[46] "Play Framework - Build Modern & Scalable Web Apps with Java and Scala." [Online]. Available: http://www.playframework.com/. [Accessed: 20-Mar-2014].

[47] "Akka." [Online]. Available: http://akka.io/. [Accessed: 13-Aug-2013].

[48] "SockJS - GitHub." [Online]. Available: https://github.com/sockjs. [Accessed: 24-Mar-2014].

[49] "Meteor Homepage." [Online]. Available: http://meteor.com/. [Accessed: 22-Jan-2013].

[50] Meteor, "Documentation - Meteor." [Online]. Available: http://docs.meteor.com/. [Accessed: 28-Aug-2013].

[51] Weswit, "About Us - Weswit." [Online]. Available: http://www.weswit.com/about. [Accessed: 11-Apr-2013].

[52] "Lighstreamer wiki." [Online]. Available: http://en.wikipedia.org/wiki/Weswit. [Accessed: 11-Aug-2013].

[53] Weswit, "Lightstreamer - Docs." [Online]. Available: http://www.lightstreamer.com/doc. [Accessed: 11-Aug-2013].

[54] "Planet Framework." [Online]. Available: http://www.planetframework.com/. [Accessed: 20-Mar-2014].

[55] XSockets, "XSockets.NET | xsockets.net." [Online]. Available: http://xsockets.net/. [Accessed: 20-Mar-2014].

Sources

[56]   R. McMurray, "Installing IIS 8 on Windows Server 2012: The Official Microsoft
        IIS Site." [Online]. Available: http://www.iis.net/learn/get-started/whats-new-
        in-iis-8/installing-iis-8-on-windows-server-2012. [Accessed: 06-Mar-2014].

[57]   Microsoft, "IIS 8.0 WebSocket Protocol Support: The Official Microsoft IIS
        Site." [Online]. Available: http://www.iis.net/learn/get-started/whats-new-in-
        iis-8/iis-80-websocket-protocol-support. [Accessed: 06-Mar-2014].

[58]   V. Sor, "Most popular application servers | Java Code Geeks." [Online].
        Available: http://www.javacodegeeks.com/2013/03/most-popular-application-
        servers.html. [Accessed: 20-Mar-2014].

[59]   "Jetty - Servlet Engine and Http Server." [Online]. Available:
        http://www.eclipse.org/jetty/. [Accessed: 06-Mar-2014].

[60]   "Apache Tomcat - Welcome!" [Online]. Available: http://tomcat.apache.org/.
        [Accessed: 06-Mar-2014].

[61]   "Planet Framework | Documentation | Deploy your Planet Application."
        [Online]. Available:
        http://www.planetframework.com/documentation/1.10/Deploying/Deploy-
        your-Planet-Application/. [Accessed: 06-Mar-2014].

[62]   Node.js, "Node.js Homepage." [Online]. Available: http://nodejs.org/.
        [Accessed: 31-Jul-2013].

[63]   G. Paul, "Beginner's Guide to Node.js (Server-side JavaScript)." [Online].
        Available: http://www.hongkiat.com/blog/node-js-server-side-javascript/.
        [Accessed: 06-Mar-2014].

[64]   "Getting Started with OWIN and Katana : The Official Microsoft ASP.NET
        Site." [Online]. Available: http://www.asp.net/aspnet/overview/owin-and-
        katana/getting-started-with-owin-and-katana. [Accessed: 06-Mar-2014].

[65]   "Windows Live Messenger - Wikipedia." [Online]. Available:
        http://en.wikipedia.org/wiki/Windows_Live_Messenger. [Accessed: 24-Mar-
        2014].

[66]   Knockout, "Knockout : Home." [Online]. Available: http://knockoutjs.com/.
        [Accessed: 24-Mar-2014].

[67]   SeleniumHQ, "Selenium WebDriver." [Online]. Available:
        http://docs.seleniumhq.org/projects/webdriver/. [Accessed: 20-Mar-2014].

[68]   LearnBoost, "Socket.IO: the cross-browser WebSocket for realtime apps."
        [Online]. Available: http://socket.io/#browser-support. [Accessed: 20-Mar-
        2014].

[69]   Weswit, "Lightstreamer - Homepage." [Online]. Available:
        http://www.lightstreamer.com/. [Accessed: 01-Apr-2014].

Sources

[70]   Play, "Java WebSockets with Play." [Online]. Available:
       http://www.playframework.com/documentation/2.1.x/JavaWebSockets.
       [Accessed: 13-Aug-2013].

[71]   Play, "JavaComet with Play." [Online]. Available:
       http://www.playframework.com/documentation/2.1.x/JavaComet. [Accessed:
       13-Aug-2013].

[72]   M. Wasson and P. Fletcher, "Dependency Injection in SignalR : The Official
       Microsoft ASP.NET Site." [Online]. Available:
       http://www.asp.net/signalr/overview/signalr-20/extensibility/dependency-
       injection. [Accessed: 20-Mar-2014].

[73]   SignalR, "SignalR on Nuget." [Online]. Available:
       https://www.nuget.org/packages/Microsoft.AspNet.SignalR/2.0.2 . [Accessed:
       25-Mar-2014].

[74]   Meteor, "Meteor Roadmap | Trello." [Online]. Available:
       https://trello.com/b/hjBDflxp/meteor-roadmap. [Accessed: 28-Aug-2013].

[75]   D. Matalon, "How To Use MySQL With Meteor." [Online]. Available:
       http://www.fastcolabs.com/3007015/how-use-mysql-meteor. [Accessed: 28-
       Aug-2013].

[76]   "Meteor on Windows." [Online]. Available: http://win.meteor.com/. [Accessed:
       28-Aug-2013].

[77]   Meteor, "GitHub commit - Meteor." [Online]. Available:
       https://github.com/meteor/meteor/commit/5b2240df4f951f95b3433572fd5d
       40c376deb038. [Accessed: 20-Apr-2014].

[78]   SockJS, "SockJS Client - GitHub." [Online]. Available:
       https://github.com/sockjs/sockjs-client.

[79]   ExpressJS, "Express - node.js web application framework." [Online]. Available:
       http://expressjs.com/. [Accessed: 04-Aug-2013].

[80]   LearnBoost, "Socket.IO - GitHub." [Online]. Available:
       https://github.com/LearnBoost/socket.io. [Accessed: 24-Mar-2014].

[81]   LearnBoost, "Socket.IO - Wiki." [Online]. Available:
       https://github.com/LearnBoost/socket.io/wiki. [Accessed: 24-Mar-2014].

[82]   D. Baulig, "socket.io and Express. Tying it all together. | blinzeln." [Online].
       Available: http://www.danielbaulig.de/socket-ioexpress/. [Accessed: 09-Mar-
       2014].

[83]   R. Price, "JavaScript Date, Time And Node.js." [Online]. Available:
       http://www.robertprice.co.uk/robblog/2011/05/javascript_date_time_and_n
       ode_js-shtml/. [Accessed: 25-Mar-2013].

Sources

[84]   LearnBoost, "Transports of Engine.IO - GitHub." [Online]. Available:
        https://github.com/LearnBoost/engine.io/tree/master/lib/transports.
        [Accessed: 24-Mar-2014].

[85]   LearnBoost, "Engine.IO commits - GitHub." [Online]. Available:
        https://github.com/LearnBoost/engine.io/commits/master. [Accessed: 24-
        Mar-2014].

[86]   LearnBoost, "Socket.IO commits - GitHub." [Online]. Available:
        https://github.com/LearnBoost/socket.io/commits/master. [Accessed: 24-
        Mar-2014].

[87]   Weswit, "Lightstreamer - Products." [Online]. Available:
        http://www.lightstreamer.com/products. [Accessed: 21-Mar-2014].

[88]   A. Alinone, "Difference between DISTINCT and MERGE mode?" [Online].
        Available: http://forums.lightstreamer.com/showthread.php?873-Difference-
        between-DISTINCT-and-MERGE-mode. [Accessed: 11-Apr-2013].

[89]   Weswit, "'Hello World' with Lightstreamer Colosseo."[Online]. Available:
        http://www.lightstreamer.com/docs/articles/JavaScript-Client_and_Java-
        Data-Adapter_Tutorial_Colosseo/index.htm. [Accessed: 11-Aug-2013].

[90]   Weswit, "Lightstreamer: General Concepts." [Online]. Available:
        http://www.lightstreamer.com/latest/Lightstreamer_Allegro-Presto-
        Vivace_5_1_Colosseo/Lightstreamer/DOCS-SDKs/General Concepts.pdf.
        [Accessed: 21-Mar-2014].

[91]   D. Krafzig, K. Banke, and D. Slama, *Enterprise SOA. Service-Oriented
        Architecture: Best Practices*. John Wait, 2005.

[92]   M. N. Huhns and M. P. Singh, "Service-Oriented Computing: Key Concepts and
        Priniciples," *Internet Computing*, vol. 9, no. 1, pp. 75–81, 2005.

[93]   AngularJS, "AngularJS — Superheroic JavaScript MVW Framework." [Online].
        Available: http://angularjs.org/. [Accessed: 25-Mar-2014].

[94]   Oracle, "ExecutorService (Java Platform SE 7)." [Online]. Available:
        http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ExecutorServi
        ce.html. [Accessed: 10-Mar-2014].

[95]   A. Alinone, "From Push Technology to the Real-Time Web." [Online].
        Available: http://www.slideshare.net/alinone/from-push-technology-to-the-
        realtime-web. [Accessed: 26-Mar-2013].

[96]   Marketwire, "Weswit Named 'Cool Vendor' by Leading Analyst Firm for its
        Lightstreamer Product," 2012. [Online]. Available:
        http://www.marketwired.com/press-release/weswit-named-cool-vendor-by-
        leading-analyst-firm-for-its-lightstreamer-product-1649111.htm. [Accessed: 21-
        Mar-2014].

Sources

[97]   Play, "Getting started with Play." [Online]. Available:
        http://www.playframework.com/documentation/2.1.x/Installing. [Accessed:
        22-Mar-2014].

[98]   Play, "Play documentation Home." [Online]. Available:
        http://www.playframework.com/documentation/2.1.x/Home. [Accessed: 22-
        Mar-2014].

[99]   Play, "Play documentation Scala Home." [Online]. Available:
        http://www.playframework.com/documentation/2.1.x/ScalaHome. [Accessed:
        22-Mar-2014].

[100]   C. Hewitt, P. Bishop, and R. Steiger, "A universal modular ACTOR formalism
        for artificial intelligence," *IJCAI'73 Proceedings of the 3rd international joint
        conference on Artificial intelligence*, pp. 235–245, 1973.

[101]   Avaje, "Avaje Ebean ORM Peristence Layer (Java) - Compare to JPA." [Online].
        Available: http://www.avaje.org/. [Accessed: 20-Apr-2014].

[102]   Play, "Play documentation: JavaTest." [Online]. Available:
        http://www.playframework.com/documentation/2.1.x/JavaTest. [Accessed:
        22-Mar-2014].

[103]   Play, "Philosophy." [Online]. Available:
        http://www.playframework.com/documentation/2.0/Philosophy. [Accessed:
        13-Aug-2013].

[104]   TIOBE, "TIOBE Software: Tiobe Index." [Online]. Available:
        http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html.
        [Accessed: 20-Aug-2013].

[105]   SignalR, "SignalR Documentation - Git." [Online]. Available:
        https://github.com/SignalR/SignalR/wiki. [Accessed: 24-Mar-2014].

[106]   "Unit Testing on SignalR Hubs with 2.0 RC1." [Online]. Available:
        http://software.intel.com/en-us/blogs/2013/09/25/unit-testing-on-signalr-
        hubs-with-20-rc1. [Accessed: 10-Mar-2014].

[107]   P. Fletcher, "SignalR Performance: The Official Microsoft ASP.NET Site."
        [Online]. Available: http://www.asp.net/signalr/overview/signalr-
        20/performance-and-scaling/signalr-performance. [Accessed: 22-Mar-2014].

[108]   "Issue 576 SignalR - GitHub." [Online]. Available:
        https://github.com/SignalR/SignalR/issues/576. [Accessed: 24-Mar-2014].

[109]   SignalR, "SignalR releases - GitHub." [Online]. Available:
        https://github.com/SignalR/SignalR/releases. [Accessed: 24-Mar-2014].

Sources

[110] J. Galloway, "Jon Galloway: Bleeding edge ASP.NET: See what is new and next for MVC, Web API, SignalR and more… on Vimeo." [Online]. Available: http://vimeo.com/68378251. [Accessed: 27-Aug-2013].

[111] Meteor, "Meteor supported platforms - GitHub." [Online]. Available: https://github.com/meteor/meteor/wiki/Supported-Platforms. [Accessed: 24-Mar-2014].

[112] A. Young, "DailyJS: Windows and Node: Getting Started." [Online]. Available: http://dailyjs.com/2012/05/03/windows-and-node-1/. [Accessed: 11-Mar-2014].

[113] Meteor, "Meteor: the screencast." [Online]. Available: https://www.meteor.com/screencast. [Accessed: 22-Mar-2014].

[114] ShiggyEnterprises, "Unit Testing Meteor Apps." [Online]. Available: http://shiggyenterprises.wordpress.com/2013/05/17/unit-testing-meteor-apps/. [Accessed: 22-Mar-2014].

[115] "Laika - Testing Framework for Meteor." [Online]. Available: http://arunoda.github.io/laika/. [Accessed: 24-Mar-2014].

[116] "WS issue in Chrome - GitHub." [Online]. Available: https://github.com/meteor/meteor/issues/1140. [Accessed: 24-Mar-2014].

[117] J. O'Dell, "Open-source Meteor takes a huge $11.2M first round." [Online]. Available: http://venturebeat.com/2012/07/25/meteor-funding/. [Accessed: 30-Aug-2013].

[118] SignalR, "Crank - GitHub." [Online]. Available: https://github.com/SignalR/SignalR/tree/dev/src/Microsoft.AspNet.SignalR.Crank. [Accessed: 24-Mar-2014].

[119] Apache, "Apache JMeter - Apache JMeter™." [Online]. Available: https://jmeter.apache.org/. [Accessed: 23-Mar-2014].

[120] Gatling, "Gatling Project - Stress Tool." [Online]. Available: http://gatling-tool.org/. [Accessed: 24-Mar-2014].

[121] "Phantomjs Issue 11018 - GitHub." [Online]. Available: https://github.com/ariya/phantomjs/issues/11018. [Accessed: 24-Mar-2014].

[122] SlimerJS, "SlimerJS." [Online]. Available: http://slimerjs.org/. [Accessed: 08-Apr-2014].

[123] HtmlUnit, "HtmlUnit - Welcome to HtmlUnit." [Online]. Available: http://htmlunit.sourceforge.net/. [Accessed: 23-Mar-2014].

Sources

[124] HtmlUnit, "HtmlUnit - Changes." [Online]. Available:
http://htmlunit.sourceforge.net/changes-report.html. [Accessed: 24-Mar-2014].

[125] Highcharts, "Highcharts - Interactive JavaScript charts for your webpage."
[Online]. Available: http://www.highcharts.com/. [Accessed: 23-Mar-2014].

[126] SignalR, "SignalR Performance - GitHub." [Online]. Available:
https://github.com/SignalR/SignalR/wiki/Performance. [Accessed: 24-Mar-2014].

[127] Microsoft, "Network Monitor." [Online]. Available:
http://technet.microsoft.com/en-us/library/cc938655.aspx. [Accessed: 23-Mar-2014].

[128] Wireshark, "Wireshark · Go Deep." [Online]. Available:
http://www.wireshark.org/. [Accessed: 23-Mar-2014].

[129] Telerik, "Fiddler - The Free Web Debugging Proxy by Telerik." [Online].
Available: http://www.telerik.com/fiddler. [Accessed: 23-Mar-2014].

[130] Microsoft, "Perfmon." [Online]. Available: http://technet.microsoft.com/en-us/library/bb490957.aspx. [Accessed: 23-Mar-2014].

[131] Google, "v8 - V8 JavaScript Engine - Google Project Hosting." [Online].
Available: https://code.google.com/p/v8/. [Accessed: 24-Mar-2014].

[132] "Programming Languages Benchmarks." [Online]. Available:
http://attractivechaos.github.io/plb/. [Accessed: 18-Mar-2014].

[133] S. Souders, "Roundup on Parallel Connections | High Performance Web Sites."
[Online]. Available:
http://www.stevesouders.com/blog/2008/03/20/roundup-on-parallel-connections/. [Accessed: 11-Mar-2014].

[134] K. Johannessen, "Byte calculations - GitHub." [Online]. Available:
https://github.com/kjohann/MasterThesis/blob/master/Loadtests/Results/Bytes sent received analysis.md.

[135] "java - ExecutorService slow multi thread performance - Stack Overflow."
[Online]. Available:
http://stackoverflow.com/questions/7155608/executorservice-slow-multi-thread-performance. [Accessed: 18-Mar-2014].

[136] W. P. Riley-Land, "Performance Test: WebSockets vs. Server Sent Events."
[Online]. Available: http://kun.io/blog/39590083022/Performance-Test:-WebSockets-vs.-Server-Sent-Events-(EventSource). [Accessed: 16-Mar-2014].

[137] "connect-sse - GitHub." [Online]. Available:
https://github.com/andrewrk/connect-sse. [Accessed: 24-Mar-2014].

Sources

[138] M. Nehlsen, "Server Sent Events vs. WebSockets." [Online]. Available: http://matthiasnehlsen.com/blog/2013/05/01/server-sent-events-vs-websockets/. [Accessed: 16-Mar-2014].

[139] "SockJS at npmjs.org." [Online]. Available: https://www.npmjs.org/package/sockjs. [Accessed: 24-Mar-2014].

[140] P. Verhas, "Java Memory Leak | Javalobby," 2013. [Online]. Available: http://java.dzone.com/articles/java-memory-leak. [Accessed: 18-Mar-2014].

[141] P. Fletcher, "Tutorial: SignalR Self-Host : The Official Microsoft ASP.NET Site." [Online]. Available: http://www.asp.net/signalr/overview/signalr-20/getting-started-with-signalr-20/tutorial-signalr-20-self-host. [Accessed: 24-Mar-2014].

[142] Xamarin, "Xamarin - Build mobile apps for iOS, Android, Mac and Windows." [Online]. Available: https://xamarin.com/. [Accessed: 01-Apr-2014].

[143] R. Thurlow, "RPC: Remote Procedure Call Protocol Specification Version 2." [Online]. Available: https://tools.ietf.org/html/rfc5531. [Accessed: 20-Mar-2014].

[144] Akka, "What is an Actor? — Akka Documentation." [Online]. Available: http://doc.akka.io/docs/akka/2.3.0/general/actors.html. [Accessed: 20-Mar-2014].

[145] "Trello." [Online]. Available: https://trello.com/. [Accessed: 24-Mar-2014].

[146] SignalR, "SignalR commits - GitHub." [Online]. Available: https://github.com/SignalR/SignalR/commits/master. [Accessed: 25-Mar-2014].

[147] "Understanding MVVM – A Guide For JavaScript Developers." [Online]. Available: http://addyosmani.com/blog/understanding-mvvm-a-guide-for-javascript-developers/. [Accessed: 25-Mar-2014].

[148] Codehouse, "Jackson JSON Processor - Home." [Online]. Available: http://jackson.codehaus.org/. [Accessed: 22-Mar-2014].

[149] "Variable Scope (JavaScript)." [Online]. Available: http://msdn.microsoft.com/en-us/library/bzt2dkta(v=vs.94).aspx. [Accessed: 22-Mar-2014].

[150] A. R. H. Girdwood, "What is a headless browser?" [Online]. Available: http://blog.arhg.net/2009/10/what-is-headless-browser.html. [Accessed: 11-Mar-2014].

Appendix

Appendix A: E-mail communication with Weswit

The following is the e-mail communication between myself and Weswit regarding settings for the Lightstreamer Server.

On Thu, Feb 13, 2014 at 5:50 PM, Simone Fabiano <simone.fabiano@lightstreamer.com> wrote:

Hello Kristian,

While I start looking at your project, could you please send me the server configuration file?
Also, can you tell me the specs of the hardware running the tests?

Thanks,
Simone.
Simone Fabiano
Software Engineer

On 2014-02-14 19:06, Alessandro Alinone wrote:

HI Kristian,

We are reproducing the tests.
We have not finished yet, and will continue on Monday, but there is already an important observation to mention.
When testing Java-based servers, it is fundamental to warm up the JVM, before collecting the results. This is the foundation of all Java-based benchmarks. So, you should make the test last much longer and discard the first few minutes of results. This should lead to more significant stats.
That being said, there are several other leverages available to tune performance, such as: GC algorithm, logging, delta delivery, JSON parser optimization, etc.
On Monday we will focus on those other aspects. On your side, if you have a chance to re-execute the test with the JVM warm up, keeping all the rest fixed, this could help.

Thanks and have a nice week-end,
Alessandro

Appendix

On Mon, Feb 17, 2014 at 1:15 PM, Kristian
Johannessen <<u>kjohann@student.matnat.uio.no</u>> wrote:

Hi!

Just thought I should let you know that running the test over a longer timeperiod
gave very improved results. After about 20 seconds, the latency stabilizes in the range
of 15-25 ms. This is about the same performance as SignalR.

Kristian

On 2014-02-17 18:43, Simone Fabiano wrote:

Hi Kristian,

I've run your tests with a few tweaks using two laptops attached to our local LAN via
cable. I tested Socket.IO and Lightstreamer to have a reference on such different
hardware.

I didn't use your chart server because I didn't have the chance to setup IIS on my
machine, so I added a simple average calculation directly on the client code. Also I
didn't limit the number of updates (I set it to 1000) and I made my code calculate the
average latency every 100 updates (after 100 updates it resets itself). Then I gathered
the average from each browser for the updates 301 to 400 and I came up with
basically the same value for both Lightstreamer (136.586ms) and Socket.IO
(136.795ms).

These are the tweaks I performed on the Lightstreamer installation

I changed VM configuration to use the G1 garbage collector and set a max pause of
5ms (don't do this unless you're using at least Java 7 as with previous VM the G1 gave
some weird results). To do that edit the LS.bat file from the bin/windows folder of the
lightstreamer installation and add `-XX:+UseG1GC -XX:MaxGCPauseMillis=5` to the `JAVA_OPTS` variable.

I disabled the delta delivery of Lightstreamer: using json updates this feature is
completely useless. In the conf/lightstreamer_conf.xml file search for the
delta_delivery option and set it N (`<delta_delivery>N</delta_delivery>`)

I disabled the log. Not sure if this had any impact at all but I decided to simplify the
case

Appendix

As I already told you I got the values after a few minutes in order to properly warm up the JVM

I set the max delay to 0 in the Lightstreamer configuration (`<max_delay_millis>0</max_delay_millis>`)

At that point I was thinking about upgrading the Java JSON parser version (there is a newer one, 2.2.3) but I preferred to skip that step.

Finally I have a couple of observation about the tests:

Limiting the test to 60 connections prevent the test to actually verify the scalability of the various servers. (obviously I'm pretty confident in the Lightreamer scalability but this is a general observation)

Using browsers to run load tests is probably not the right choice as you're adding a lot of overhead to each client. We usually prefer to use java clients when we have to run our load tests. (currently our own java library does not support websockets, and I'm not sure about socket.io java clients, anyway I wrote a very simple java client for a test once. It's not general purpose but it can easily be adapted I suppose. I also suppose you've not enough time for this :-) anyway see https://github.com/Weswit/Lightstreamer-toolkit-socket.io-benchmark/tree/master/client/src/loadtestclient/client)

My final note is about the messages you use to calculate the latency: I see that each client calculates the latency only considering the messages sent by itself. This may theoretically give you mixed results based on how a server implements its logic: imagine what happens if a server always send a message back to the client that originated it first and only after starts to send the same updates to all the other clients. (Noite that I didn't change that for my calculations)

HTH

Good Luck with your Thesis!

Simone.

--
Simone Fabiano
Software Engineer
Weswit :: www.weswit.com
Lightstreamer :: www.lightstreamer.com

E: simone.fabiano@lightstreamer.com
Skype: w.simone.fabiano

Appendix

Lightstreamer, selected Cool Vendor by Gartner, is the most advanced server for real-time bi-directional data delivery over the Web. More Than Just WebSockets...

Hi!

I just tried to do a couple of long running test (4 minutes) with the tweaks you suggested. It does not seem to improve the results I got yesterday when I ran my tests longer (but with no other tweaks except from the max_delay_millis setting set to 0). There are minimums as low as 9ms and peaks as high as 30ms, but the average is a little under 20 (which is around the same as SignalR).

So, maybe the tweaks have no effect on my system (since it doesn't have a lot of performance), or the low load makes it take no effect. Anyways, I will bring your results into my discussion regarding the results. This discussion will also cover the observations you made about the tests (I have thought of the same things).

I can give some insight into why my tests are as they are:

60 connections was not the plan initially, but limitations with using browsers made that the maximum (uses almost 7 GB of my 8GB of RAM with 30 browsers). My initial idea was to use a headless browser, but I cannot use Phantom. It does not support WebSockets and it would probably have the same behavior as Chrome. With Chrome, most servers refuse any connections after 6 (HTTP), even from different browsers. There is Slimer, which is Gecko based, but this is not fully headless and not mature enough for me to find it reliable. Therefore I landed on Firefox driven by Selenium - not optimal, but it gives some results at least.

Another reason to use browsers, is that I get a common testbed for all frameworks and all transports. I am not only comparing the performance of the different frameworks you see. The more interesting question in my thesis is whether or not WebSockets outperforms HTTP for various uses (real-time and "traditional" request/response sites like for instance Wikipedia). I'm basically trying to see if WebSockets can be the basis for the next generation of HTTP (sort of). Using Java clients for this purpose (or other languages) seemed to force me to use different implementations of the clients for both framework AND transport. With browsers, there are just one file different on the client for each framework.

Still, in retrospect, I probably would have done it with Java clients (or something else) if I could have done my work all over again. This load testing is just the second half om my thesis, the other was a more subjective look at how the frameworks were to work with. If I could go back, I probably would have focused solely on load testing.

Regarding the average calculation: Yes, I am aware of that the method has its flaws, but it felt like the best solution. It measures how long it takes for a client to get a

Appendix

response to its own message. If some servers send back to the originating client first and others last (isn't the first alternative most logical?), that is a possible source of error that will be discussed in my thesis.

Thanks a lot for your help! I've learned a lot from this - especially regarding JVM warmup :)

Regards

Kristian Johannessen

Appendix B: All standard charts

This appendix contains all charts that can be derived directly from the raw data. Some of the graphs in the thesis are made from combining this standard data. These graphs are not included in this appendix.

Messages sent from clients

WebSockets


Server-Sent Events

Appendix

HTTP-Streaming


Long-polling

Appendix

Polling

Messages sent from clients

Messages received by server

WebSockets

Messages received by server
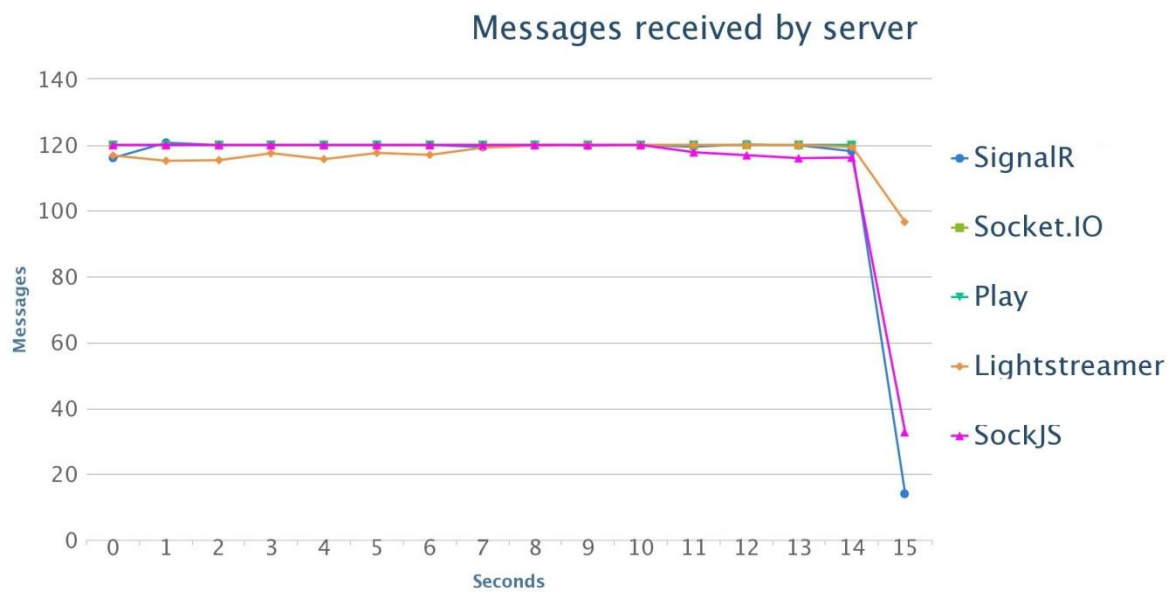
Appendix

Server-Sent Events

Messages received by server — SignalR

HTTP-Streaming

Messages received by server — Lightstreamer, SockJS
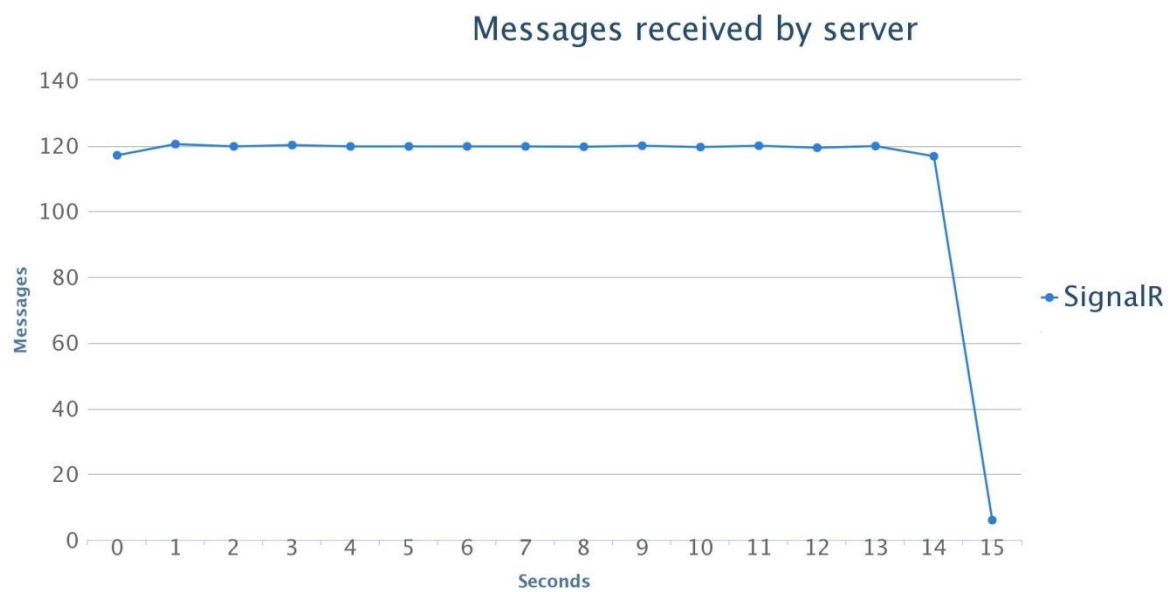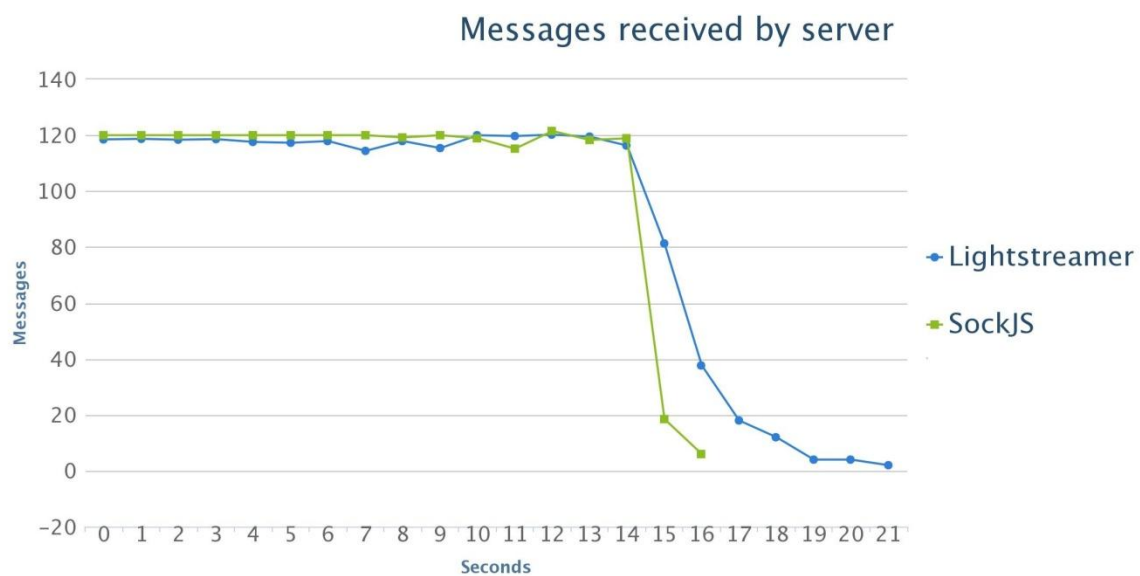
Appendix

Long-polling

Messages received by server

Polling

Messages received by server

Appendix

Messages sent from server

WebSockets

Messages sent from server pr. second

Server-Sent Events

Messages sent from server pr. second
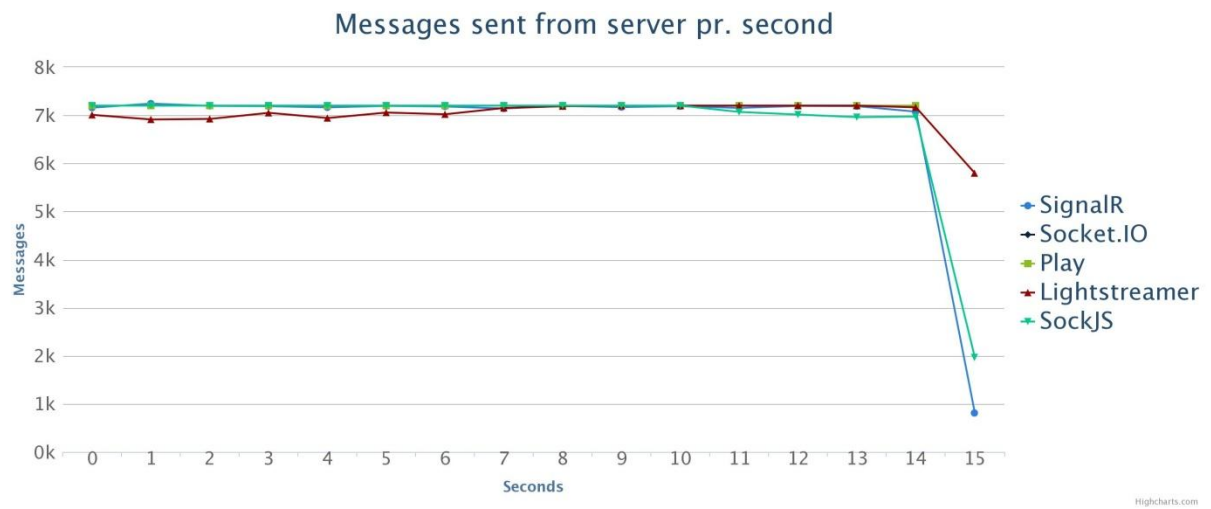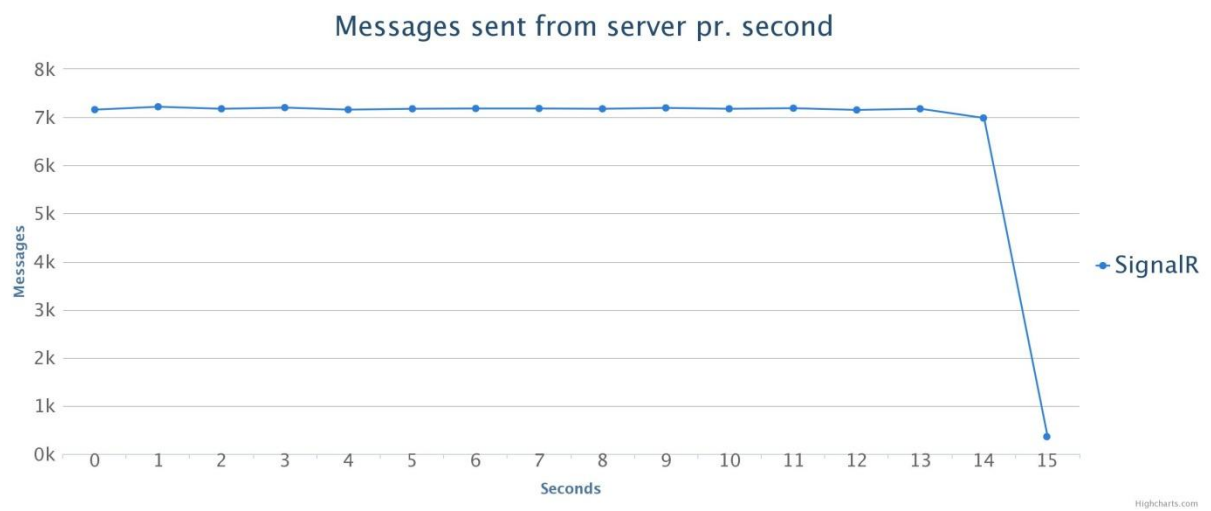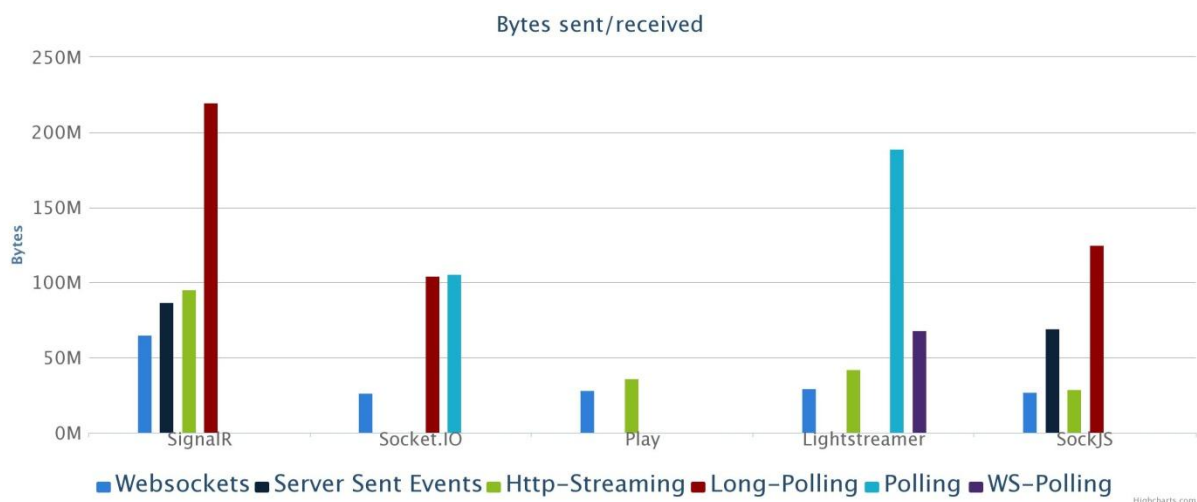
Appendix
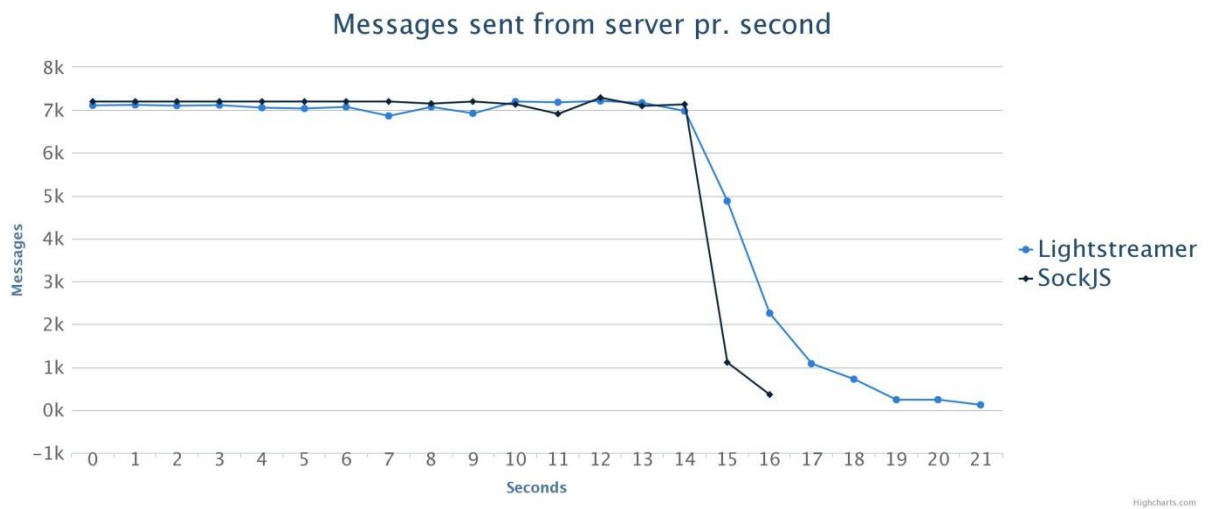
HTTP-Streaming
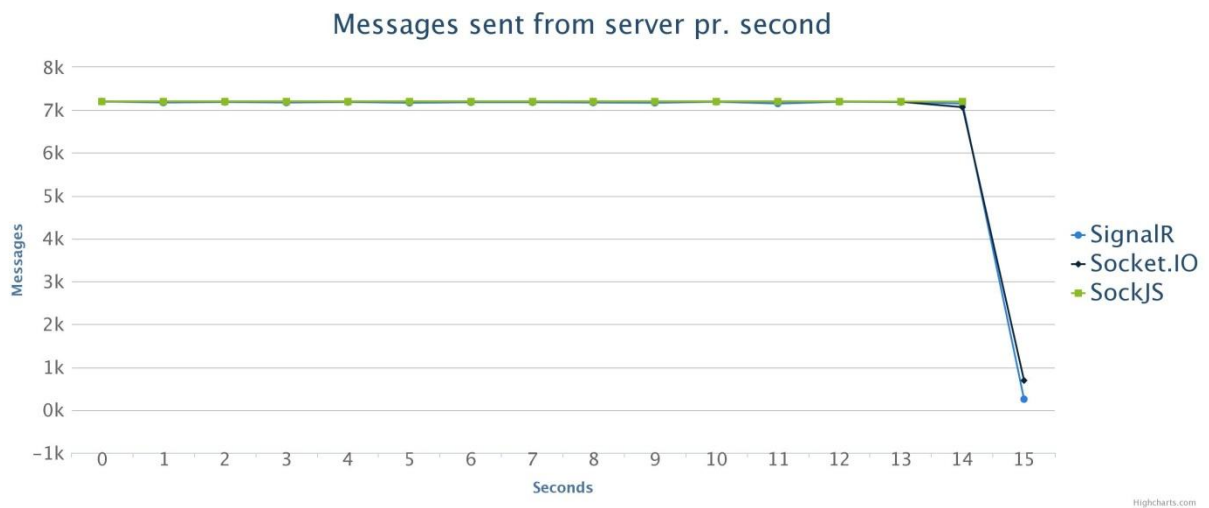
Long-polling
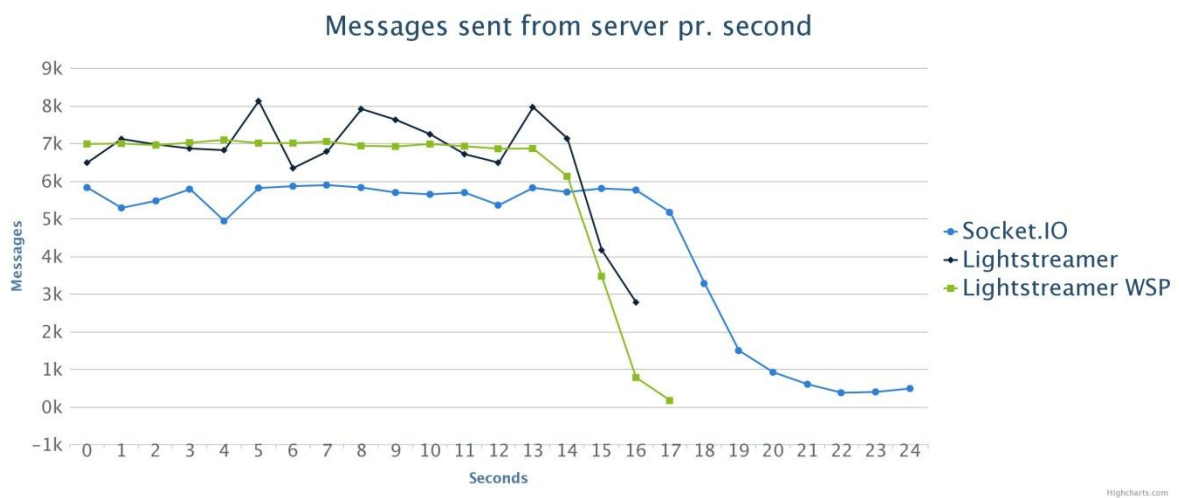
Polling

Average latency

WebSockets
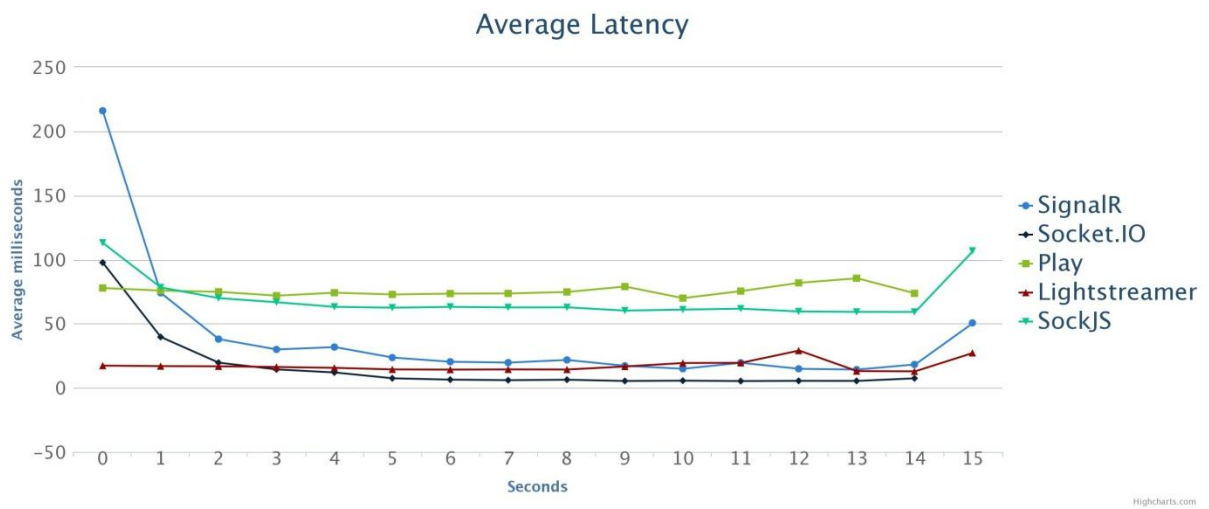
Server-Sent Events
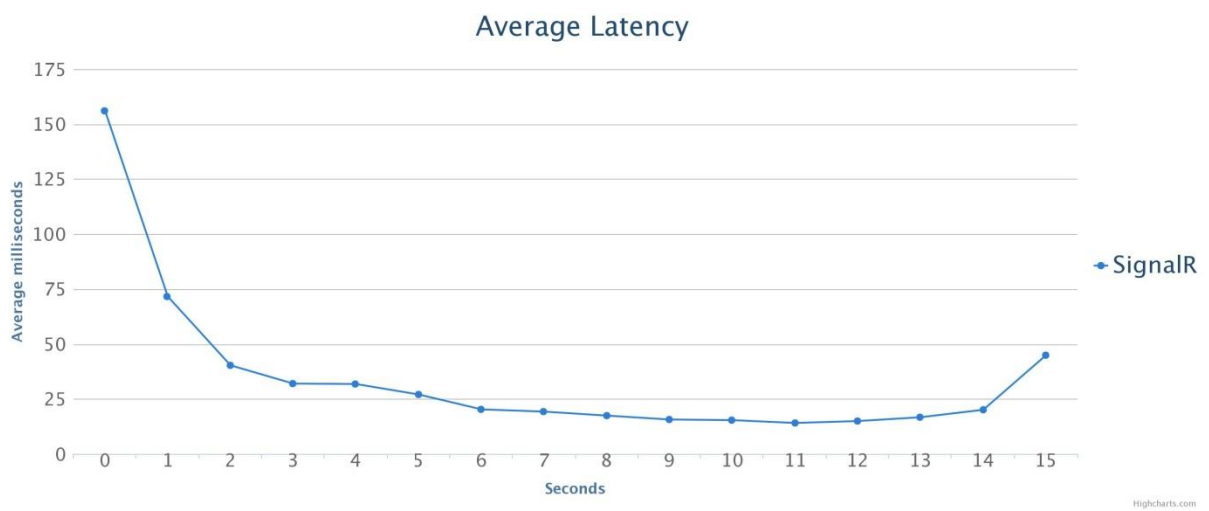
Appendix

HTTP-Streaming
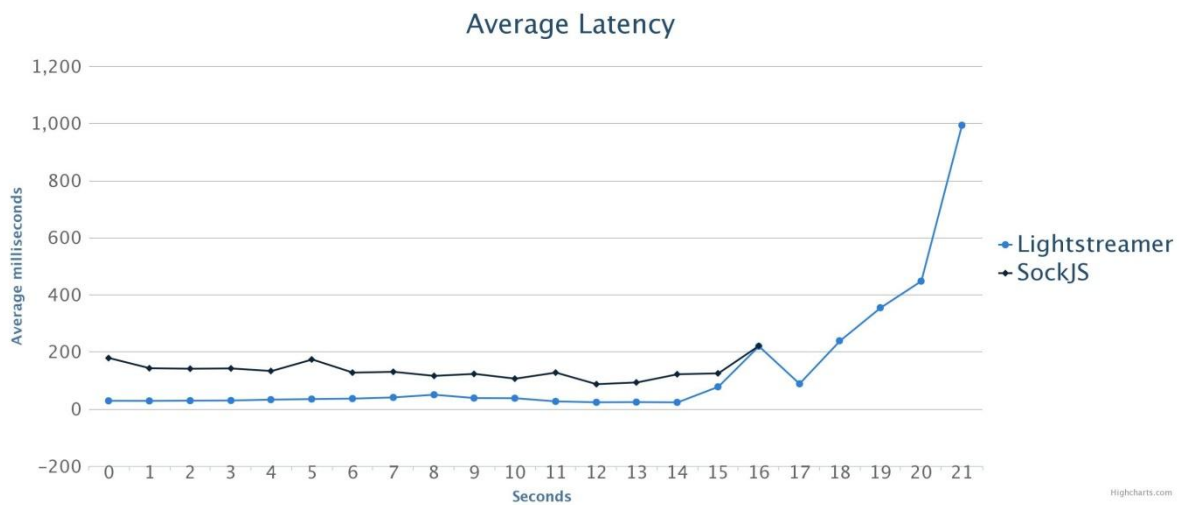
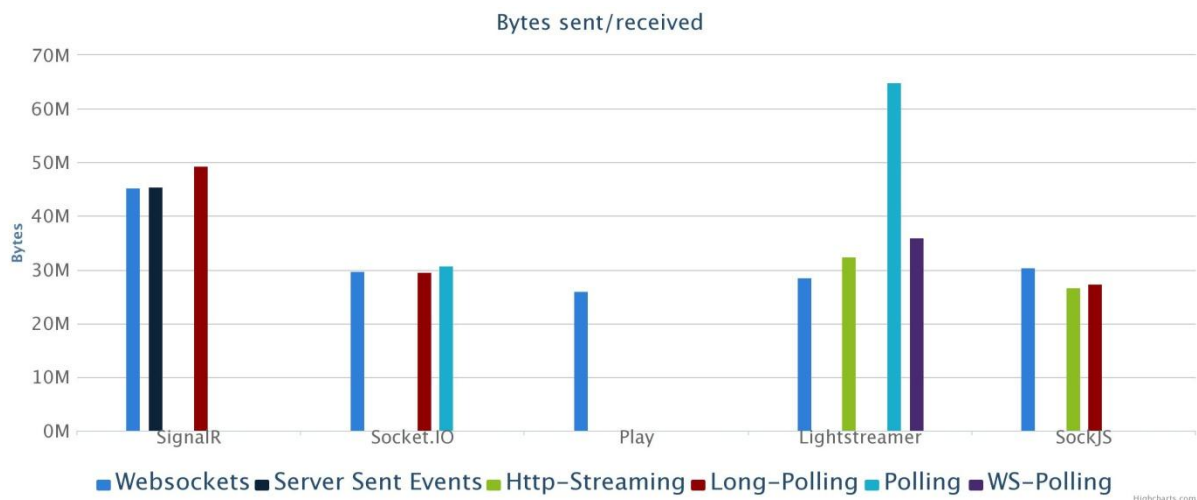## Long-polling
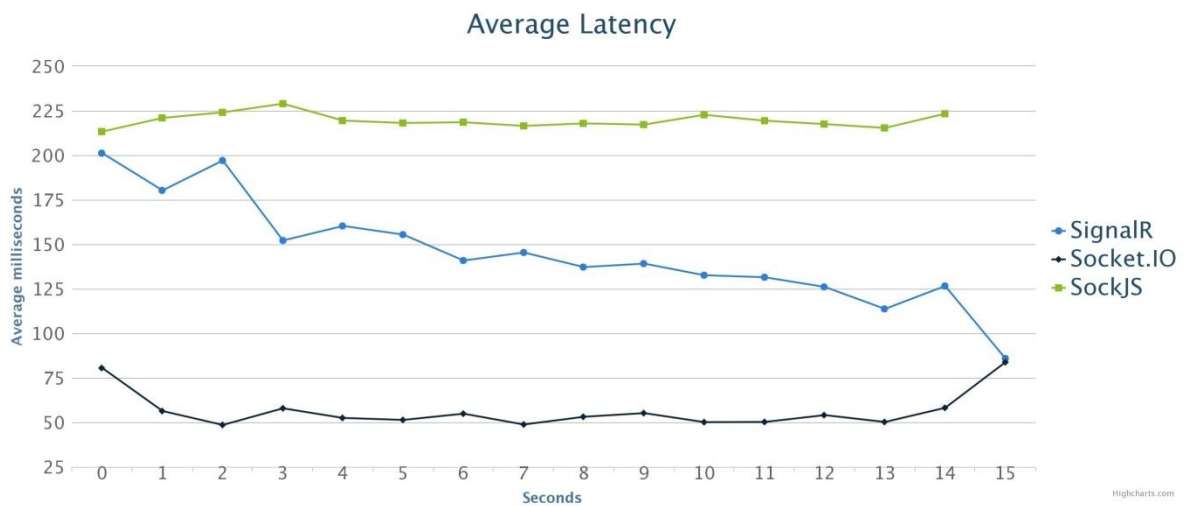
Polling
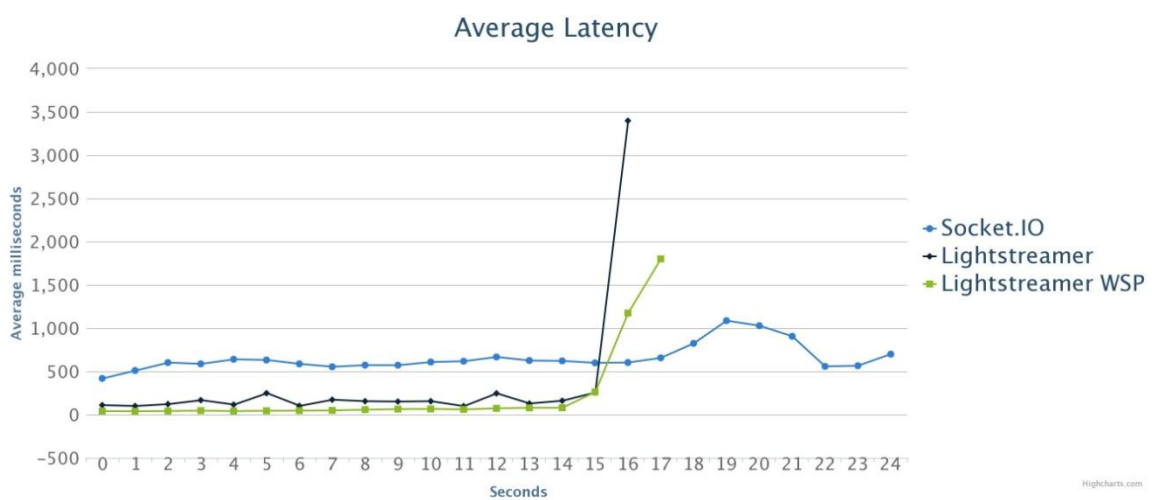
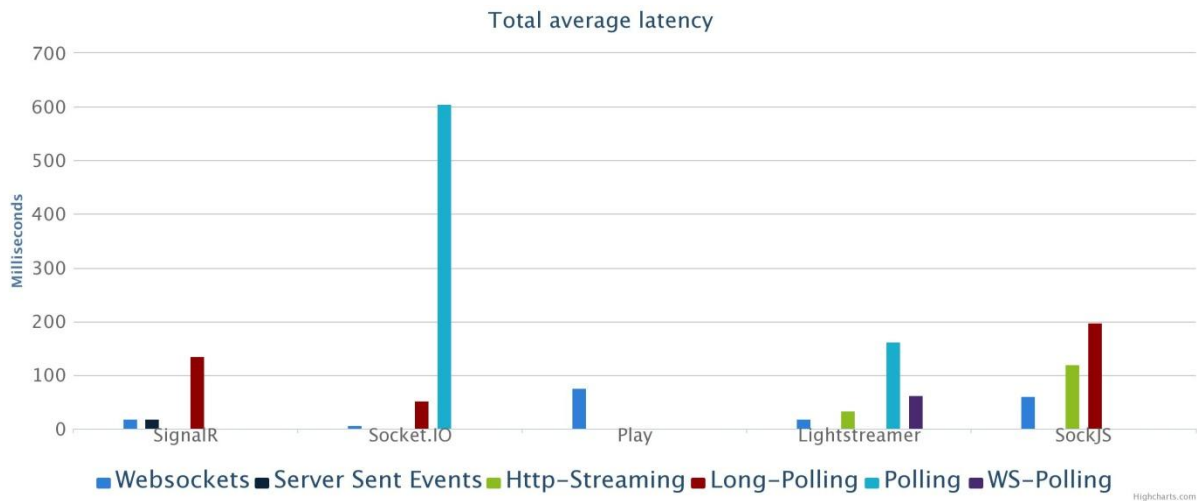# Median Processor usage
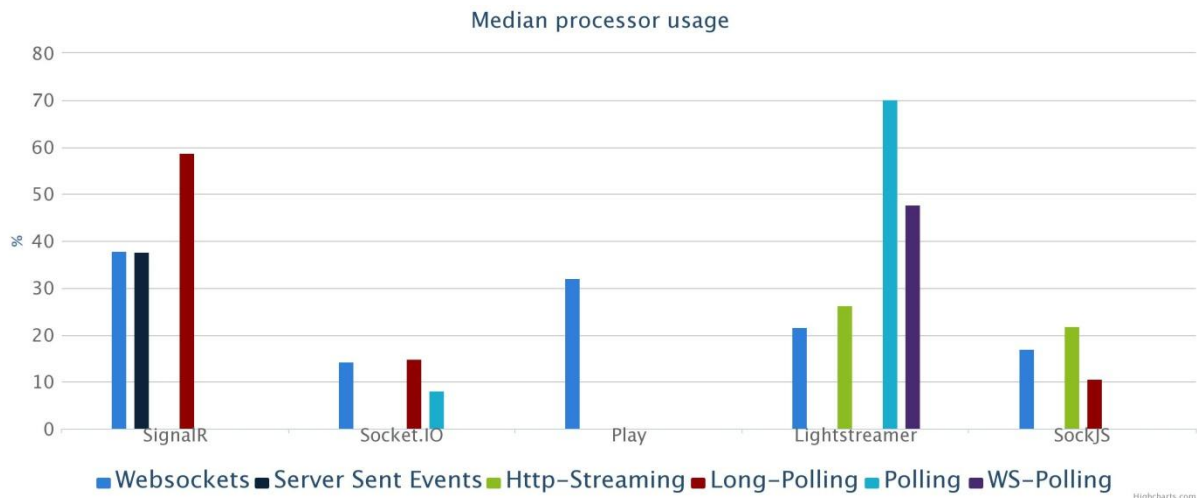
Max memory consumption
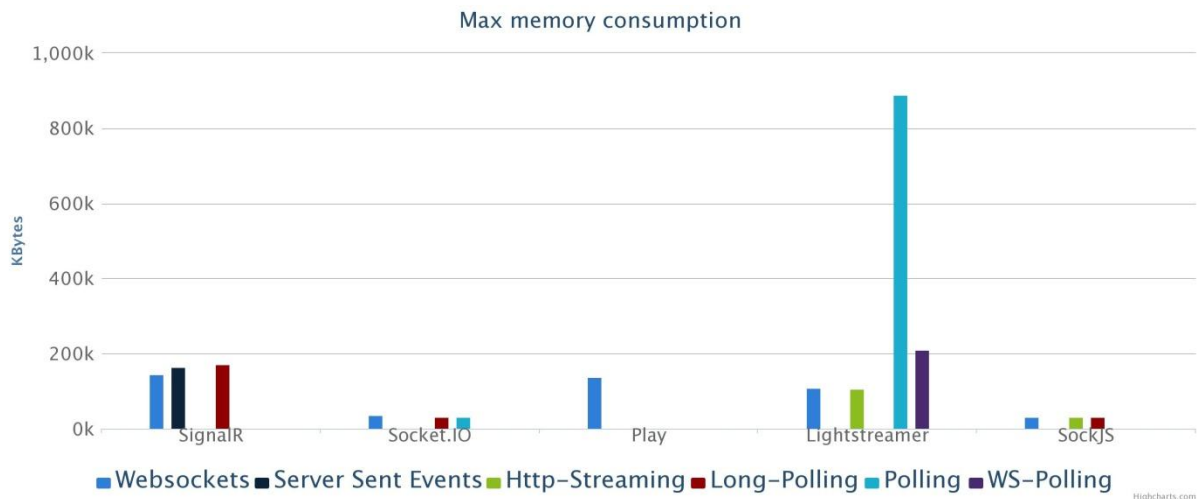
Bytes sent/received (calculated)

Bytes sent/received (captured)

Total average latency (original)

Total average latency (fixed for readability)