NB: All in-paper notes will be marked with the word TODO in upper case letters.
TODO: Frontpage goes here! Fix in a pdf creator?

Abstract

Contents

Abstract i
List of figures vii
Preface viii
Introduction 1
1.1 Problem statement 2
1.2 Terminology 3
1.3 Outline 3
Background 4
2 The World Wide Web 5
2.1 HTTP/1.0 5
2.2 HTTP/1.1 6
2.3 HTTP/2.0 6
3 Real-time 7
3.1 The Real-time Web with HTTP 8
3.1.1 Polling 8
3.1.2 Long-polling 9
3.1.3 HTTP-Streaming 10
3.1.4 Comet 11
3.1.5 Server-Sent Events 11
3.2 WebSockets 12
3.2.1 How it works 12
3.2.2 The WebSockets API 13
3.3 Drawbacks of HTTP techniques 14
3.3.1 Really real time? 14
3.3.2 When long-polling becomes polling 15
3.3.3 Streaming techniques 16
3.4 HTTP was never designed for real time 16
3.4.1 Overhead 16
3.4.2 Unidirectional 17
3.5 WebSockets is still young 18
3.5.1 Know when to use it 18
3.5.2 Know how to use it 18
3.6 The use of real time 19

3.7 Conclusion 20
4 Frameworks for real time web applications 21
4.1 SignalR 21
4.2 Socket.IO 21
4.3 Atmosphere 21
4.4 Sails.js 22
4.5 Play! Framework 22
4.6 SockJS 22
4.7 Meteor 22
4.8 Lightstreamer 22
4.9 Planet Framework 23
4.10 XSockets.NET 23
5 Servers 23
5.1 .NET 23
5.2 Java 24
5.3 JavaScript 24
5.4 Writing your own 24
Project part 1: Hands on development 25
6 Methodology 26
6.1 Selection criteria 26
6.1.1 WebSockets support 26
6.1.2 Fallback support 26
6.1.3 Presentation 26
6.1.4 Maturity 26
6.2 Description of test application 27
6.3 Discussion of use cases 27
6.4 Evaluation 27
6.4.1 Documentation 28
6.4.2 Maintainability 28
6.4.3 Simplicity 28
6.4.4 Browser support 29
6.4.5 Maturity 29
6.4.6 Other criteria 29
6.5 Limitations 29

6.5.1 Cloud based solutions 30
6.5.2 Rapid development 30
6.6 Other choices 30
6.6.1 Common UI 30
6.6.2 Choice of database engine 30
6.6.3 Functional testing 30
6.7 Selected frameworks 31
6.7.1 Socket.IO 31
6.7.2 Lightstreamer 31
6.7.3 Play Framework 32
6.7.4 SignalR 32
6.7.5 Meteor 33
7 Results 34
7.1 Socket.IO 34
7.1.1 Documentation 34
7.1.2 Simplicity 35
7.1.3 Maintainability 36
7.1.4 Browser support 37
7.1.5 Maturity 37
7.2 Lightstreamer 38
7.2.1 Documentation 38
7.2.2 Simplicity 39
7.2.3 Maintainability 42
7.2.4 Browser support 43
7.2.5 Maturity 43
7.3 Play Framework 44
7.3.1 Documentation 44
7.3.2 Simplicity 44
7.3.3 Maintainability 48
7.3.4 Browser support 48
7.3.5 Maturity 49
7.4 SignalR 49
7.4.1 Documentation 49
7.4.2 Simplicity 50

7.4.3 Maintainability 52
7.4.4 Browser support 53
7.4.5 Maturity 53
7.5 Meteor 53
7.5.1 Documentation 53
7.5.2 Simplicity 55
7.5.3 Maintainability 56
7.5.4 Browser support 58
7.5.5 Maturity 58
Project part 2: Load testing 59
8 Methodology 60
8.1 Test scenario 60
8.2 Test data 61
8.3 Test setup 61
8.4 Choice of setup 62
8.5 Number of runs 63
8.6 Displaying data 63
8.7 Configurations 64
8.8 Monitoring network traffic 64
8.9 Monitoring of processor 64
8.10 Monitoring of memory usage 65
8.11 Different servers and platforms 65
8.12 Use case of test setup 65
8.13 Limitations 66
8.13.1 Meteor 66
8.13.2 Using browsers 66
8.13.3 Network capture 66
8.13.4 Streaming with Play Framework 67
8.14 Testing idle connections resource usage 67
8.15 Raw data 67
9 Results 67
9.1 Messages sent from clients 68
9.2 Messages received by server 69
9.3 Messages sent from server 70

9.4 Average latency 71
9.4.1 WebSockets 71
9.4.2 Server Sent Events 72
9.4.3 Http-Streaming 72
9.4.4 Long-Polling 72
9.4.5 Polling 73
9.5 Median processor usage 74
9.6 Maximum memory usage 74
9.7 Bytes sent/received 75
9.8 Idle clients with WebSockets 76
10 Analysis 76
10.1 Message frequency 76
10.2 Average latency 80
10.2.1 Frameworks 80
10.2.2 Transports effect on latency 81
10.3 Machine resources 83
10.4 Transports effect on network traffic 84
10.5 WebSockets idle connections resource usage 86
Conclusion 88
11 Frameworks 89
12 WebSockets or HTTP? 89
13 Further work 90
Sources 91
Appendix 93
Appendix A 94

List of figures

Preface

Introduction

The concept of real time 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 chapter.

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

Getting some hands on experience with different frameworks will also help 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. I intend to find out if you need a framework no matter the size of 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 eachother. This is the most interesting aspect, 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 performance of these and of methods for real time. To handle this, the thesis is separated into two parts. The first part covers the development aspect, while the other handles performance.

Both parts are summarized in the conclusion chapter. 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:

What framework do I consider to be the best for real time web applications?

No matter the functionality you want to build, you always want the best tools to do it. When it comes to software though, this question will have a nuanced answer. Some frameworks may handle different aspects better than other, while others may have the best performance.

Is it always useful to use a framework?

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.

Does WebSockets outperform the old, established HTTP methods for real time?

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

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 performance wise, it will not be the foundation for the next generation. However, if it does, can it change the way browsers communicate with browsers in more aspects than real time?

1.2 Terminology

TODO: Define framework == library in some cases (socket.io, SignalR)
TODO: Transports == WebSockets, SSE, Long-Polling....
TODO: WebSockets is not plural, websockets are.
TODO: IntelliSense
TODO: Object orienting
TODO: Functional languages

1.3 Outline

Background

2The World Wide Web

The World Wide Web has been available for 20 years (TODO: History of the world wide web (1)), and is still considered a young technology. But over those 20 years it has changed in almost every thinkable way.

Over the years, 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, Cascading Style Sheets (CSS) has given web pages a more vivid look with various styling options. Asynchronous JavaScript and XML (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 meet one of the newest aspect of web browsing, namely real time applications. A real time application is when clients receive updates from the server as they occur. (TODO: crossref more info section). Real-time web applications has been around for some time, but before they have relied on the aging HTTP 1.1 protocol.

2.1 HTTP/1.0

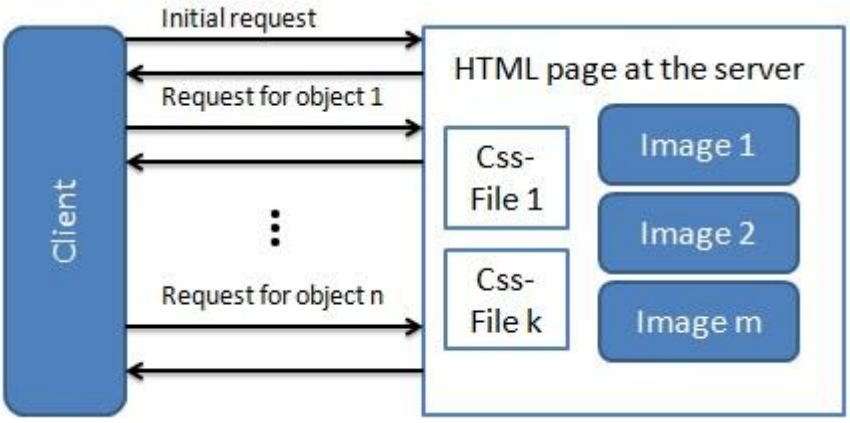
Version 1.0 of HTTP was created in 1996-the World Wide Web's childhood (TODO: (6): http 1.0). Besides text, web pages maybe had a few embedded objects at that time[1]. But as the Internet grew, it soon became clear that the user experience had to be improved.

At this time, around the mid 90s, CSS too was in its childhood (TODO: (7): Css saga). 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.

Downloading one element in a HTML-file, or even the HTML-file itself from the server required one TCP request (TODO: figure (2.1)). The server then replied and closed the connection. Throughout the duration of the request, the client waited. (TODO (9): key differences). Getting a HTML-file with a style sheet and three images

[1] Embedded objects consisted mostly of images, but also some early forms of style sheets.
[2] 17902 vs. 57915 words.

then required five consecutive requests in total.



2.2 HTTP/1.1

After just three years, HTTP/1.1 was released as a standard. It introduced several improvements. One of these was persistent connections. This allowed several request to made at the same time (TODO: (8): Network performance http 1.1). 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 (TODO: (4): http 1.1), letting servers start sending a response without knowing how long it was. In theory, it could be infinite, as we shall see in section (TODO: crossref (3.3)).

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 (TODO: (9): Key differences) 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.

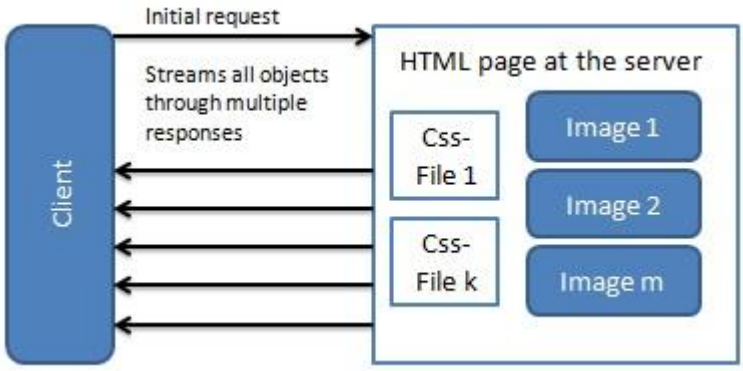
Updating from version 1.0 to 1.1 may not seem like a giant leap, but it actually was. Looking at the lengths of the different protocol specifications is an indication of how much more detailed the 1.1 protocol is[2].

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 (TODO: source) in 2012. This was then used as the basis for the HTTP/2.0 specification in November that year (TODO: source, first version). The draft has had steady development throughout 2013, and it is planned to be delivered as a proposed standard November 2014 (TODO: source).

[2] 17902 vs. 57915 words.

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. (TODO: source). With this concept, a server can provide all embedded objects in an HTML page as several responses to a single request. (TODO: figure).



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. (TODO: source). It will be exiting to follow the development of the draft.

3Real-time

As mentioned in (TODO: crossref 1), one of the newest additions to the World Wide Web is real time applications. 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 (TODO: figure (2.2)).

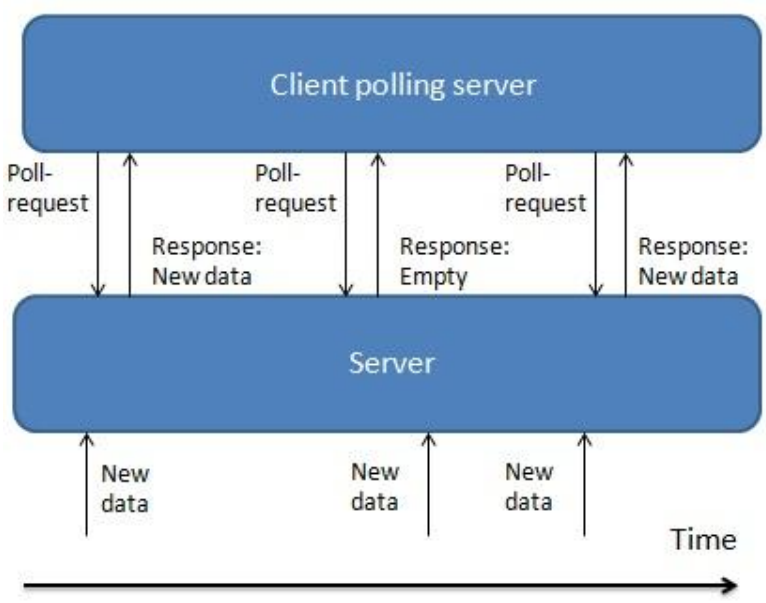
"As soon as it happens" is exactly what real time is: providing updates for the client immediately, without the need for refreshing the page on the client side. And as the examples above show, the real time aspect of an application can be either a small feature, or the core concept of the application.

3.1 The Real-time Web with HTTP

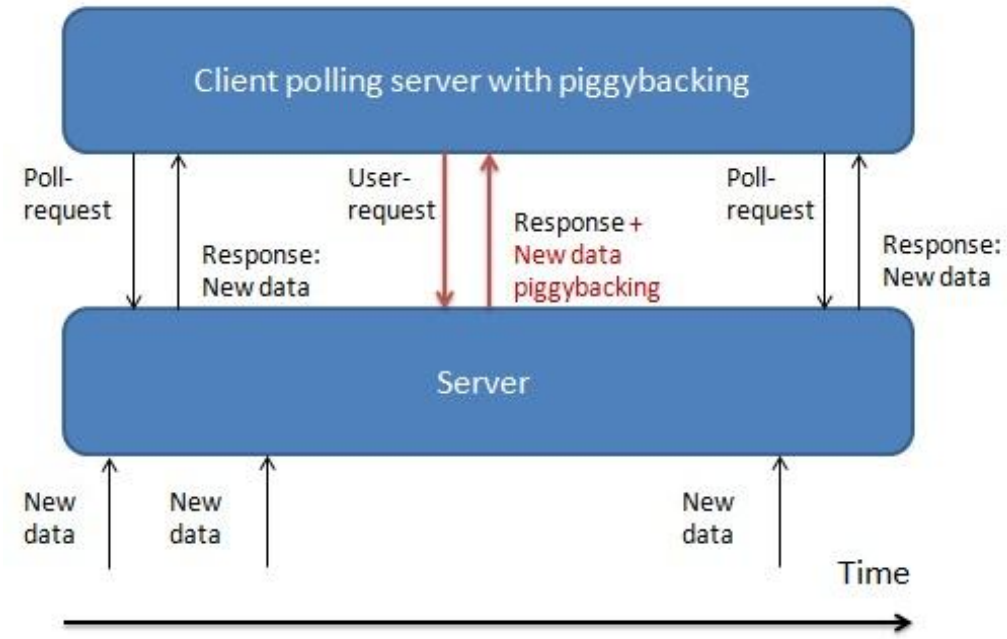
Recently the concept of real time web has become a buzzword. Real time works by pushing updates to clients as they appear. This is not how HTTP works though–the client always has to initiate the communication. To accommodate the growing need for applications of this sort, several techniques have been 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 (TODO: (10): Pro Html5). The server then instantly sends back a response–either containing new data or just an empty response if there was nothing to retrieve (TODO: figure 3-1). 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.



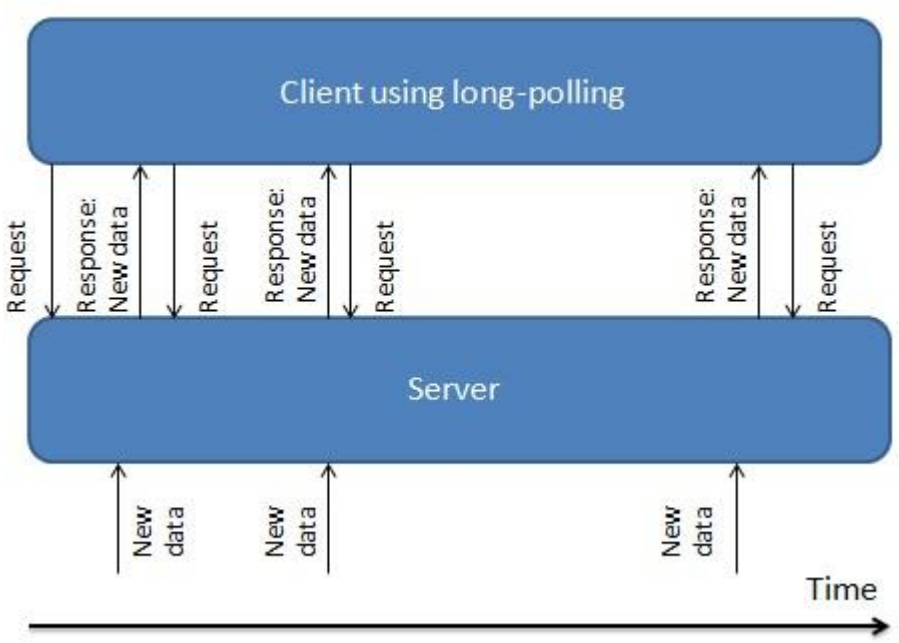
There is a way to improve a little upon polling: piggybacking (TODO: (11): Comet and reverse AJAX). Polling the server at regular intervals is usually done in parallel to other HTTP-requests initiated by client actions. These actions, of course, also get responses back from the server. Piggybacking takes advantage of this by also sending updated data back via the response. In that way, the client may get new data between the polling interval (TODO: figure 3-2).



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 response (TODO: (11): Comet and reverse AJAX). This allows the server hold the response back. It

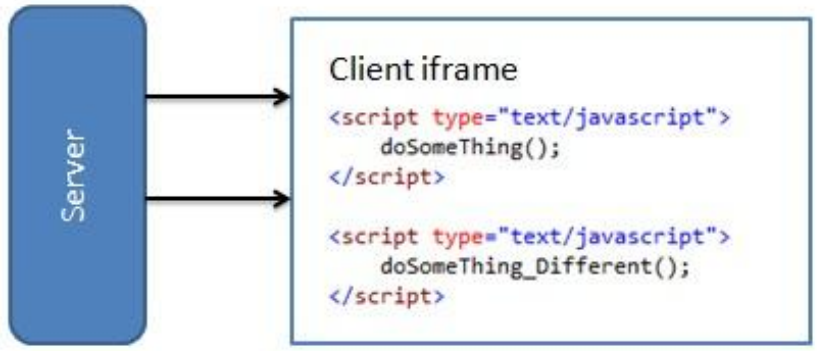
cannot do this forever though, so eventually it times out. The client then makes a new request. (TODO: figure 3-3). (TODO: (12): A comparison push/pull)



3.1.3 HTTP-Streaming

HTTP streaming is an old technique introduced by Netscape as early as 1992 - well before even HTTP 1.0 became standard. (TODO: (12): A comparison push/pull). Two forms of streaming exist, namely page streaming and service streaming. The first of the two has 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. This type of streaming is otherwise known as XHR-streaming. (TODO: find source). 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.

The most common implementation of this technique today is the so-called forever frame. As mentioned in section (TODO crossref background http1.1), HTTP 1.1 allows a server to send a response without knowing in advance its length. A forever frame is just an iframe that receives script-tags in an everlasting response from a server. (TODO: (13): The foreverframe tech). Browsers execute script-tags when they it reads them. The server therefore sends data to the clients wrapped up as such (TODO: figure).



3.1.4 Comet

Long-Polling and HTTP Streaming are often referred to as Comet or Comet Programming (TODO: (14): Comet: low latency). 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 (TODO: (15): Bayeux protocol), but it has yet to become approved by the IETF as a RFC[3]. With the introduction of WebSockets, it may never be.

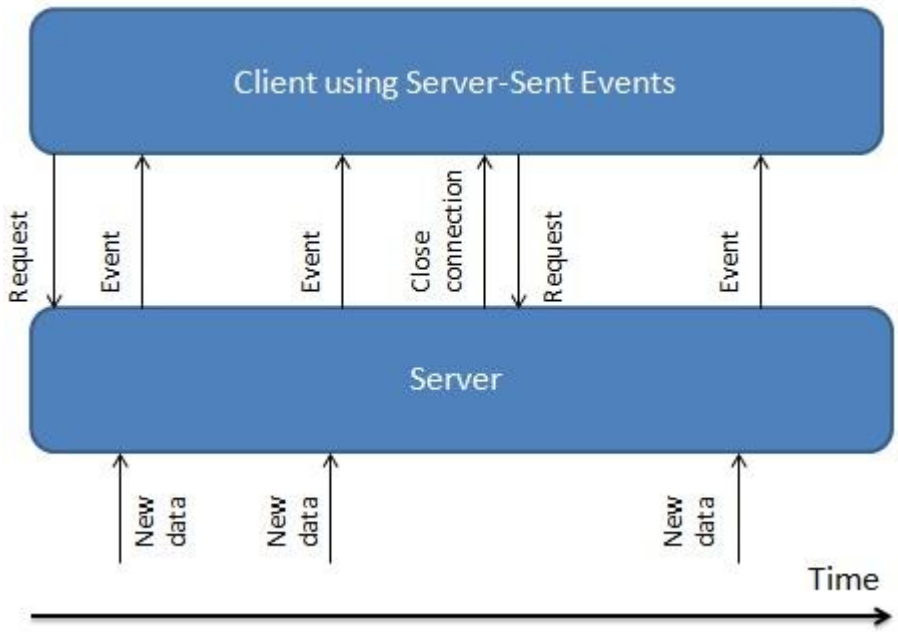
3.1.5 Server-Sent Events

Let's move on into the borders of Web 2.0 with HTML5s Server-Sent Events. (TODO: (16): Html5 server push part 1). Server-Sent Events takes advantage of the "text/event-stream" Content Type of HTML5 to push messages to the client. (TODO: (17): Stream updates with..). 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. But 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. (TODO: (17): Stream updates with..). Server-Sent Events behavior is quite close to long-polling (TODO: figure 3-5).

Unlike long-polling, though, developers using Server-Sent Events have access to a simple API. (TODO: (18): Server Sent Events). The API gives access to the EventSource interface, which provides straightforward JavaScript code. It allows the server-side to fire events in the browser and, in turn, update the content on the client-side. 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.

[3] Internet Engineering Task Force - Request for Comment series: see http://www.rfc-editor.org/



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 TODO: http1.1 in background about upgrade request-header), but just now a new version is in the works. With WebSockets developers can now finally make use of the upgrade request-header.

In December 2011, the WebSockets protocol became a proposed Internet Engineering Task Force (IETF) specification[4].(TODO: (19): WebSockets becomes). 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 bidirectional communication technologies that use HTTP as a transport layer to benefit from existing infrastructure"* (TODO: (20): WS protocol, section 1.1)

3.2.1 How it works

WebSockets, like HTTP, use TCP as underlying protocol. While HTTP needs several "hacks" (TODO: crossref real time http), WebSockets provides bidirectional communication right out of the box.

The WebSocket protocol use the same ports as HTTP and HTTPS (80 and 443, respectively). This allows the initial handshake to be done via traditional HTTP (TODO: figure 4-1). The client states that it wants to use WebSockets, and the server

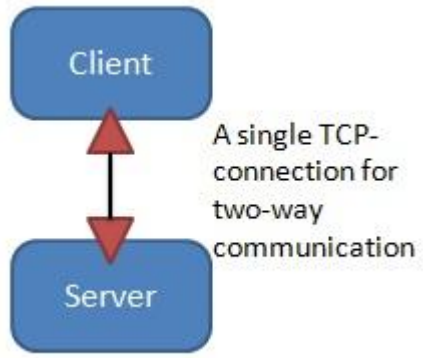
[4] RFC6455 (TODO: source) is the document with the specification.

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=9925811.1029531631.1358508580.1359736844.1
  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
```

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 (TODO: (10): Pro Html5), while WebSockets sends messages in frames with only two bytes overhead (TODO: (21): About WS). Frames can be sent both ways at the same time eliminating the need for more than one request at the same time (TODO: figure 4-2).



3.2.2 The WebSockets API

As with Server-Sent Events, WebSockets has its own API (TODO: (22): WS API), 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 for 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."* (TODO: (20): WS protocol, section 1.5)

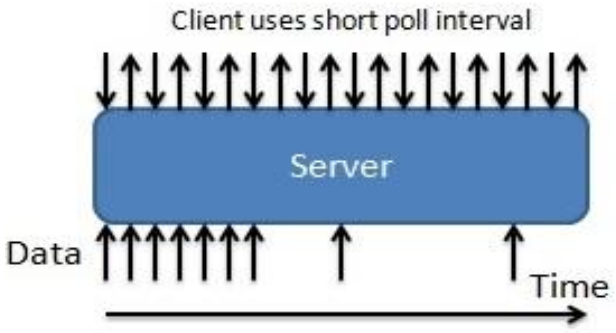
3.3 Drawbacks of HTTP techniques

In section (TODO: real time http), I gave a rudimentary description of different ways to achieve real time, communication with HTTP. They mostly work in the same way, but uses some different settings for keeping connections open and pushing messages to the client. Most used is probably long-polling, because even the oldest browsers support it. However, there are also some issues that the following sections will describe.

3.3.1 Really real time?

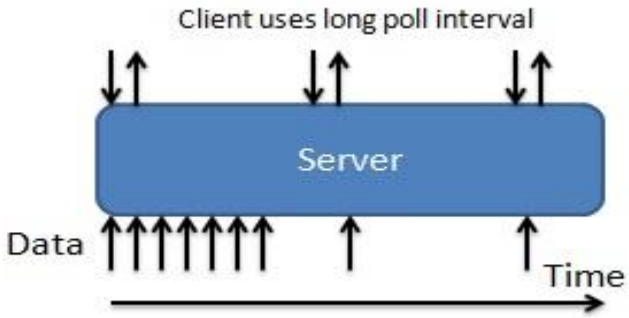
Long-polling builds upon the idea of polling. But whereas polling is a naïve approach, long-polling is smarter. One of the major issues with normal polling is how to determine the polling interval.

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. With frequent updates, this would provide a lot of stress on 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. (TODO: figure 5-1).



With a longer interval, 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

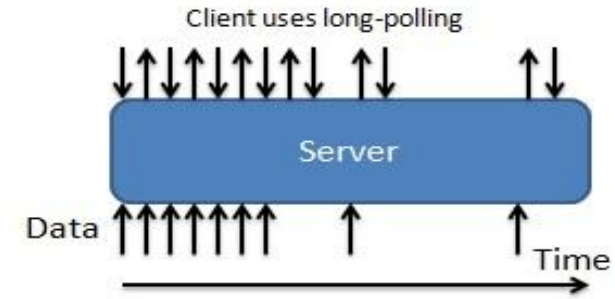
good choice for such scenarios. A weather application for instance, might get new updates every hour. This client can use this knowledge to poll the server accordingly.



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 respond at once. (TODO: (10): Pro Html5). This effect makes long-polling work just as regular polling at a short interval. Comparing (TODO: figure 5-1) to (TODO: figure 5-3), one can see that long-polling does not outperform polling as long as the server-side updates are frequent.



Norges Bank Investment Management[5] provides a counter on their homepage that shows the total value of the Norwegian Government Pension Fund. 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, fakes real time as it polls the server every 30 seconds and gets the values from the past 30 seconds.

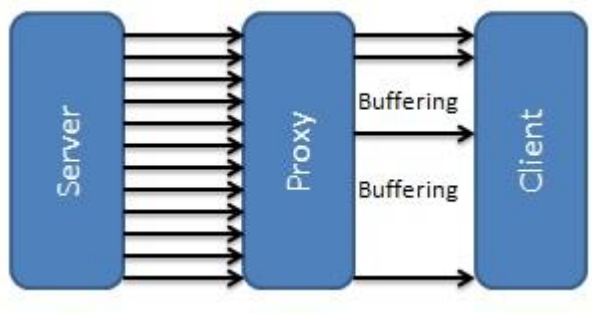
[5] http://www.nbim.no – TODO: counter is no longer there...

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. You can configure Server Sent Events to work almost like long-polling, but it uses push just as streaming does. (TODO: sse section).

A forever frame allows the server to push updates to the client wrapped up in script-tags, but it is far from perfect. Client-side there has to be some extra handling 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. (TODO: (10): Pro Html5). The nature of the HTTP-protocol may cause these to buffer the response, thus creating a lot of latency for the client. (TODO: figure 5-4). To avoid this, many Comet-based streaming solutions fall back to long-polling if it detects buffering.



A forever frame makes the developer write some extra code to handle the incoming scripts. The EvenSource interface gives developers a more powerful toolbox for handling incoming events. (see section (TODO: sse section)). Utilizing pure eventhandlers also ensures that there is no need for cleaning up after the incoming data. But, in the end, Server Sent Events are still HTTP. And as we shall see, the 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. (TODO: background http 1.1). 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.

3.4.1 Overhead

In section (TODO: crossref How it works), I mentioned that headers in HTTP requests/responses can accumulate to hundreds of bytes. (TODO: (10): Pro Html5). Peter Lubbers and Frank Greco made a simple application for comparing polling to

WebSockets. (TODO: (23): Benefits of WS). To shed some light over 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 just 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. With just a few clients this is not a lot. But when you have hundreds of thousands 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[6]. 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[7].

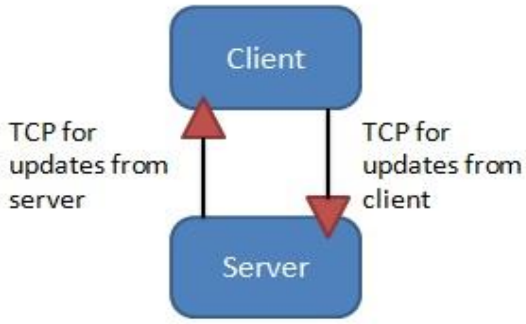
Using polling to represent HTTP against WebSockets is a little unfair in my opinion. Polling is the naïve approach of achieving real time, while WebSockets is designed for it. However, it shows my point: HTTP-headers have much excess data, but most of the time 99% of this data is irrelevant. Achieving 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's 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 (TODO: (20): WS protocol)–HTTP isn't. In fact, no matter how you look at it, or how you try to hack, HTTP remains unidirectional.

As a result of this, most real time applications with HTTP have to use several TCP-connections. (TODO: figure 5-5). 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. (TODO :Network performance http 1.1). 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.

[6] 87 100 000 bytes * 8 = 696 800 000 bits / 1024^2 = 665 Mbits
[7] 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. As mentioned in section (TODO: crossref How it works), 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% (TODO: (24): w3Schools) of the browser market with IE8 and IE9 as the most dominant (TODO: (24): w3Schools). None of these supports 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 and Socket.IO 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 fall-back provided by the framework doesn't suffice.

3.5.1 Know when to use it

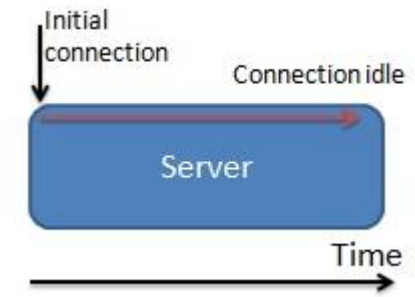
Writing an application with some real time elements is a different task than writing a full-blown dynamic, real time application. Examples of the two is an online newspaper with a live comment-section and a chat room, respectively.

In the first case using WebSockets may provide more pain than gain. Updates in a comment section does 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 users' 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 never

benefit from using WebSockets. You get less overhead in request-headers, but your application will have to serve mostly idle connections. The only real server to client communication is when the client request a new page (TODO: figure 5-6).



Understanding your application's environment is another vital aspect. WebSockets should handle proxies and firewalls gracefully. (TODO: (10): Pro Html5). 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 (TODO: (25): How Ws interact proxies), 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

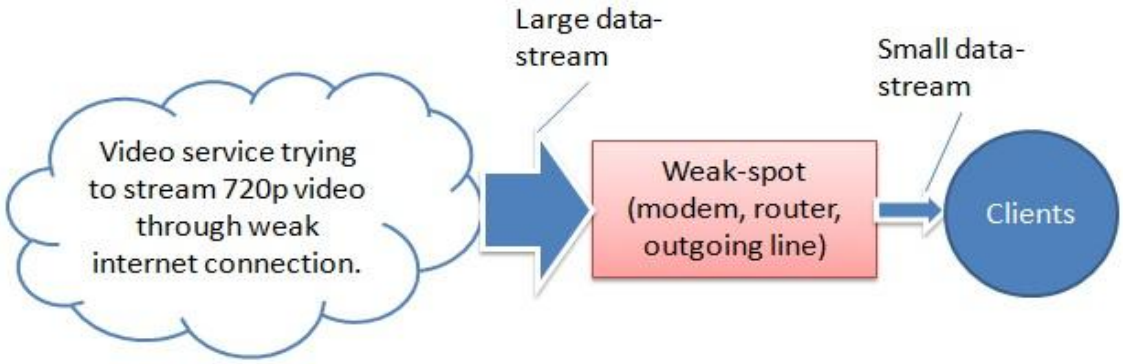
The World Wide Web has seen many innovations throughout its lifespan. Each time something new comes around, it is hard to determine if it has come to stay. It is always a question of need: Do we really need this? Is it useful to me as a consumer? Real-time is no different from any other new developments; 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 bid to all users. Forcing their clients to refresh a web page manually to see the latest bid, would render it completely useless. On the other side of the scale we find web sites that utilizes 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 users' 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 mostly about the users' 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. (TODO: figure 5-7). 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 (TODO: (10): Pro Html5), will play a central part in future improvements to video streams.



3.7 Conclusion

We have seen that WebSockets is superior to HTTP when it comes to bidirectional communication. Sometimes, though, 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 as 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.

I believe that in the future, when WebSockets has been around for a long while, it 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 because it provides vast, and yet unseen, possibilities.

Finally, my initial problem was the question of WebSockets's position in the future of the World Wide Web. Do I believe it is the future? Well, the answer is both yes and no. Yes because it is the future for bi-directional communication applications. It will

render HTTP mostly unused for the purpose as soon as the issue of backwards has vanished. Still, HTTP will remain king of the hill in “traditional” web applications that rely on requesting content in a unidirectional manner.

4 Frameworks for real time web applications

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 10 that, in my opinion, is the most widespread. Five of these will be selected for further studies.

4.1 SignalR

SignalR (TODO: ref asp.net/signalr) is a library for the .NET framework that makes it easy for developers to add real time functionality to applications. It has clients for both web applications, Windows Phone, ordinary Windows Store apps, other Windows applications and even Android an iPhone apps[8].

The library provides to 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 (TODO: server wiki git). Hubs introduces even more abstraction on top of Persistent Connections. These utilize a RPC (Remote Procedure Call)[9] API that allows for simple real time programming.

4.2 Socket.IO

Socket.IO is a library for Node.js that *“aims to make realtime apps possible in every browser and mobile”* (TODO: from socket.io). Since it runs on Node, it only uses JavaScript as a language. Thus, the available clients are browsers, either mobile or desktop.

Using an event driven architecture, it closely resembles the WebSocket API. Socket.IO extends upon this though, by letting developers name their own events as well as using the traditional WebSocket events (TODO: crossref).

4.3 Atmosphere

The Atmosphere framework (TODO: git page) is a framework built for Java servers. It uses a JavaScript client that supports all major browsers (TODO: tutorial.html at async io). It provides an API that is more low level than the Hubs API of SignalR, but a little higher than the Persistent Connections.

---

[8] In order to write C# code on Android and iPhone, one has to use the cross platform framework Xamarin (https://xamarin.com/).

[9] A remote procedure call is when a client calls a method situated on a server directly. For more information see the specifications (TODO: rfc1057 and 5531).

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

4.4 Sails.js

Sails.js (TODO: source) is a MVC (Model View Controller) framework for Node. It has full real time capabilities through the use of Socket.IO. The way this is accomplished, is rather interesting. Through what they call "transport agnostic routing" (TODO: what is sails), 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 (TODO: playframework). It is built on Akka, a framework that uses the Actor Model[10] to help developers make distributed applications.

Play is real time enabled, meaning that it has constructs to help developers make real time functionality. These constructs though are a lot closer to bear 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 API. This object can be used to connect to a server. Server side, there is a Node.js implementation as well servers for Erlang and Python (TODO: github page). 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 application drastically.

Real time is the heart of Meteor, and SockJS is used in order to enable this. All code with Meteor is written in JavaScript, but interesting enough, it is not a Node.js framework (even though it runs on Node behind the scenes(TODO: ref docs)).

4.8 Lightstreamer

Weswit (TODO: about weswit) is the Italian company behind the oldest real time framework I've been able to find (started in 2000 according to Wikipedia (TODO: source, maybe find a better one)). Lightstreamer is the name of the framework, and from the looks of it, it is also the most comprehensive.

---

[10] An Actor is a construct used to form hierarchies of entities that handle concurrency through messaging (TODO: http://doc.akka.io/docs/akka/2.3.0/general/actors.html).

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 (TODO: docs).

4.9 Planet Framework

Planet is a real time framework for Python. It is built with an emphasis on the development of social real time websites (TODO: tour). The framework also provides its own IDE, specially design APIs and scaling mechanisms.

4.10XSockets.NET

XSockets is an event driven framework for .NET that adopts a model that somewhat resembles that of Socket.IO on the client (TODO: docs/server api). Server side it uses an approach that looks like the controllers of ASP.NET MVC.

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

5Servers

All web applications has to be hosted on a server. Various servers exist for different languages. In this chapter, I will describe some web servers that are relevant for the thesis.

5.1 .NET

Most applications for ASP.NET uses IIS (Internet Information Services) as an application server. The current version of IIS is IIS8 which was released alongside Windows 8 and Windows Server 2012 (TODO: http://www.iis.net/learn/get-started/whats-new-in-iis-8/installing-iis-8-on-windows-server-2012). With it came support for the WebSockets protocol (TODO: http://www.iis.net/learn/get-started/whats-new-in-iis-8/iis-80-websocket-protocol-support). IIS is free as it is a part of the operating system. 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 .NET framework and Visual Studio, developers get access to IIS Express. As with all products in Microsoft's Express series[11], it is a lightweight version of the original. It provides what you normally need while developing, but its performance is not enough for production.

[11] Visual Studio Express is one of these products.

5.2 Java

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

Jetty is an open source server hosted by the Eclipse foundation (TODO: http://www.eclipse.org/jetty/). It is considered a very lightweight server, 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. The latter is used as Play Framework's internal web server (TODO: SO post).

Tomcat is an open source server from Apache (TODO: tomcat.apache.com) made for Java applications. Another server from Apache, the Apache Web Server, is used by the Planet Framework as preferred deployment server (TODO: deploying planet app).

5.3 JavaScript

Node.js (Node for short) 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(TODO: Nodejs.org). A web server is a fast, scalable network application. In other words, Node itself isn't a web server, but it provides a simple mechanism for writing one (TODO: beginner Node).

5.4 Writing your own

There is nothing wrong with building your own web server. 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 ().

```
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();
        }
    }
}
```

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

TODO: What tense to write in??
TODO: Have a list over used technologies and reference that instead of footnotes everywhere?

TODO: Also write about the discussions I will have about http vs ws?

I will select and compare five different frameworks for real time web application development. In this part of the thesis, the focus will be from a programmer's perspective. I will therefore implement a simple web application, which is described in section (TODO: crossref).

The selection of frameworks made in this part, will be the foundation for the second part, where each will undergo extensive load tests–comparing each framework's performance (read more in TODO: part 2).

6.1 Selection criteria

As described in section (TODO: background), several frameworks for real time applications exist. Many have similar programming interfaces and features, but for the purpose of this thesis, only a few will be selected. Each selected framework must support the criteria listed in this section. A detailed justification of the selected frameworks is given in section (TODO: reference type cross!).

6.1.1 WebSockets support

The framework must support WebSockets. Each framework will undergo load testing (TODO: ref part 2) where the individual transport mechanisms will be compared as well as framework performance. In order to answer whether or not WebSockets is an improvement to HTTP (TODO: Problem statement), it must be one of the framework's possible transports.

6.1.2 Fallback support

For the same reasons as to why WebSockets has to be supported, at least one HTTP-transport has to be 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 will make a considerable difference on the negative scale.

6.1.4 Maturity

Real-time applications has been around for many years (TODO: Windows Live 1999), but most of the frameworks for real time web applications are a lot younger. Maturity will still be given consideration, but if a frameworks offers something unique and potentially revolutionary, it may still be selected.

6.2 Description of test application

An auction house will be 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 lead bid.
- Users must be able to place bids on all items, including their own.
- If the framework does not specify a specific template engine or other means of creating views, the application will utilize a common view implemented in Knockout[12].
- 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.

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 frameworks 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. Consequently, it will be the most decisive aspect when evaluating a framework.

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 no further, nor less, complexity, I regard it as unnecessary to test this aspect in this thesis.

6.4 Evaluation

Evaluating each framework will be done during and after the development of the test application described in section (TODO). The evaluation will be from a programmers 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 will follow a preset list of

[12] www.knockoutjs.com

criteria described below. The term “user experience” in the following subsections refers to a programmers experience.

6.4.1 Documentation

It is probably the most visited page in a programmers 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 is wasted looking for reference in the documentation.

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

6.4.2 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 though, 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 (TODO: write about testing terms in terminology). Normally though, one prefers to have both unit- and integration tests as well as other forms of testing as well (for instance functional). I will measure maintainability based on how the natural structure of the application is with each given framework and how that impacts testability.

6.4.3 Simplicity

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

Serialization an 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 (TODO: 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 will be 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 one has to write complex or tiresome code just to send a message, the client handling can be as foolproof as it wants–it does not

make the overall experience any better. I will therefore also look at what constructs are offered by the frameworks to send messages to clients.

6.4.4 Browser support

This criteria is directly linked with what protocols a framework offers. WebSockets is not supported by older browsers, nor are some of the fallbacks. How a frameworks 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 though[13]. For most use cases, this is not a demand, but considering the work I will do in part 2 (TODO: crossref), it is important to me.

6.4.5 Maturity

A lot of real time frameworks are a work in progress and has 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 uses a framework. If nobody is using it, there is often a reason. Furthermore, one can look at the amounts of bugs and errors that appear during development. A lot of errors directly related to the framework's core, is often a sign of immature code. It 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 in can make a difference for some. Especially regarding support for debugging and code analysis. I believe that these things does not matter if a framework is the best tool for a job.
- 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.

6.5 Limitations

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

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

6.5.1 Cloud based solutions

If my work only consisted of comparing the usability of different frameworks, cloud based solutions could easily 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 has become one of the "buzz-words". An example of this is the Java-framework "Atmosphere"[14].

6.6 Other choices

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

6.6.1 Common UI

The application that is going to be implemented is the same for all frameworks. It will look the same, and the functionality will be the same. To keep it as simple as possible, I will strive to share as much as the user interface code as possible. For that purpose, I have created a common user interface using Knockout that will be used when possible. If the framework under test comes bundled with some other way of creating the user interface, it will be used instead.

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 will build has a database component. To keep this as similar as possible, MySql will be used if it is supported. If possible, all application will work against the same database.

6.6.3 Functional testing

As each application will appear the same to a browser, I will use a common functional test case. This is possible as long as the user interface is similar and all frameworks use the same database table. The tests will use Selenium WebDriver to drive real browsers through a series of tests. Java will be used as implementation language for the tests.

[14] https://github.com/Atmosphere/atmosphere

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. This section describes why each of the five were selected.

6.7.1 Socket.IO

Node.js 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 chose at least one Node.js based framework.

Several libraries and frameworks exist for real time with Node.js. (TODO: crossref background). Of these, Socket.IO stands out from the crowd. It seems to have the largest commnity, and it gets a lot of attention. (TODO: some videos perhaps?)

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

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

- Adobe Flash Socket, which uses Flash to establish a TCP connection between server and client.
- Ajax long-polling
- Ajax multipart streaming (Xhr-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[15]

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. Seeing how many years of experience plays out against a lot of the new ideas of the more modern frameworks, will be interesting.

---

[15] Examples of use: Trello, an online cooperation and project planning tool (https://trello.com). Sails.js, a web application framework with real-time at its core (TODO: crossref)

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, it is obviously a trusted and 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.

But before I delve into the testing of the framework itself, I need to make a disclaimer. I have used the free license of Lightstreamer for the tests in this chapter. The free license does not come with all features included[16]. In part 2 (TODO: crossref), I used a full version of Lightstreamer.

6.7.3 Play Framework

One of the questions I seek to answer is whether you need a framework at all to implement scalable real time applications. (TODO: crossref prob stat). Play offers some help in the matter, but not a lot. It is as close to working with bare metal real time as you get.

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[17].

As a developer, you get two helper classes for real time functionality. One for WebSockets (TODO: WS api doc) and one for Comet (Http-streaming, TODO: Comet API).

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

6.7.4 SignalR

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

Two developers on the ASP.NET team started the work on SignalR in 2011 (TODO: ref commit page). The fact that Microsoft supports the project is thus no surprise.

---

[16] See full list of features here: http://www.lightstreamer.com/products
[17] See Play's homepage at the bottom: http://www.playframework.com/

SignalR builds on concepts that are familiar to .NET developers, like the use of IOC (Inversion of Control) containers (TODO: source) and Nuget (TODO: source?).

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. I really sparked my interest. Without this experience, I would probably have written my master thesis with some other topic.

6.7.5 Meteor

Meteor is not a completed framwork. 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 similiar framework. I uses JavaScript both on the client and server, which is possible through the use of Node. It is not a Node.js framework though. Meteor uses a Node.js conainer to run its server. (TODO: source docs).

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

Another interesting feature is how real time is closely integrated in the core of the framework. 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 (TODO: roadmap). I will implement the test application using MongoDB. There exists an unofficial add-on that allows you to use MySql, but I will not utilize this.

Windows is currently not supported by Meteor. An unofficial fork[18] of the project exists for Windows (TODO: win.meteor). I will use this fork for my test. The only real difference is the command line tool that has to be compatible with the Windows command line. All source code remains the same (TODO: link repo).

[18] 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.

Meteor uses SockJS (TODO: git) under the hood to perform its real time updates. SockJS received support for WebSockets only a couple of days before I started my work with Meteor. If WebSockets is not supported, SockJS falls back gracefully to one of the following transports (TODO: git client):

- Server Sent Event (Opera only)
- Http-streaming (iFrame or XHR depending on the browser)
- Long-polling
- Polling (just for really old browsers)

7 Results

This chapter gives an in-depth description of my experience with each of the frameworks. Each framework start with an introduction describing some specific choices made for it. 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 chapter (TODO).

7.1 Socket.IO

As most examples with Socket.IO shows it in conjunction with Express, I used this in my test application. Express is a web application framework (TODO: source). The only aspects of the application that uses Express is the server itself, as well as serving static files. The Socket.IO application makes use of the common user interface described in section (TODO).

7.1.1 Documentation

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

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

The structure isn't perfect. Some pieces reside in the readme (on GitHub, TODO: link), while you find other pieces in the wiki. 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 though, which leads me to believe that most of the API is set already before 1.0.

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 benefitial.

7.1.2 Simplicity

In the spirit of Node[19], Socket.IO is nimble and 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 (TODO: source wiki). It is actually oblivious to sessions, leaving it up to the server library you use to handle this (TODO: sessions with express).

Some frameworks and libraries offers 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 (TODO: source wiki). 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[20] has functionality to show current readers of article, as well as real time comment sections. Socket.IO allows developers to register different channels, or "rooms". This allows implementation of such functionality to be 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 (TODO: example).

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,
```

You have to be cautious about sending Date objects though. A common problem with JSON (TODO: source), is that Date objects don't deserialize too well.

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 (TODO: example). .

---

[19] According to Node.js's homepage (TODO: source), its event-driven model makes it lightweight and efficient.

[20] www.ign.com/?setccpref=US (American IGN).

Send to caller:

```
socket.emit('registerItemResponse', itemno);
```

Send to all:

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

Because of this simple abstraction, sending a message never require 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 of course.

A final note on features offered by Socket.IO is that it closely resembles the WebSocket API. While this API only has three 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 trickier. The event driven architecture conseals all logic regarding sending and receiving within callbacks. Luckily, there are ways to work around this.

One option is to separate all code within the callback to its own module (TODO: example). 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);
        }
    });
});
```

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. (TODO: example). 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 it 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);
```

Modules can be challenging to keep small enough. Larger modules, just as larger 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 this is no longer the case (TODO: source).

7.1.5 Maturity

As mentioned before (TODO: crossref 6.1.1), Socket.IO has not reached version 1.0 yet. GitHub claims that 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. (TODO: commit pages).

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 completly 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.js itself though. It too hasn't reached a 1.0 version, 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 is about Socket.IO though, 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

For the test application I used the free license Lightstreamer offers (TODO: licensing). 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. This application does not make use of the common user interface described in section (TODO).

7.2.1 Documentation

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

With a lot of documentation comes a great responsibility to organize it. Weswit (TODO: mentioned in section..) 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[21]. I didn't get a good sense of what the difference between these were before I found a forum post that explained it. (TODO: forum post).

Another shortcoming is the almost complete lack of tutorials - only one exist. (TODO: tutorial). That would have been ok if the samples were well documented and written, but this isn't so. Many samples accompany (TODO: figure) the framework, and these

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

illustrate different uses. Understanding them at a conceptual level is easy enough, but when you start digging into the code, trouble starts.

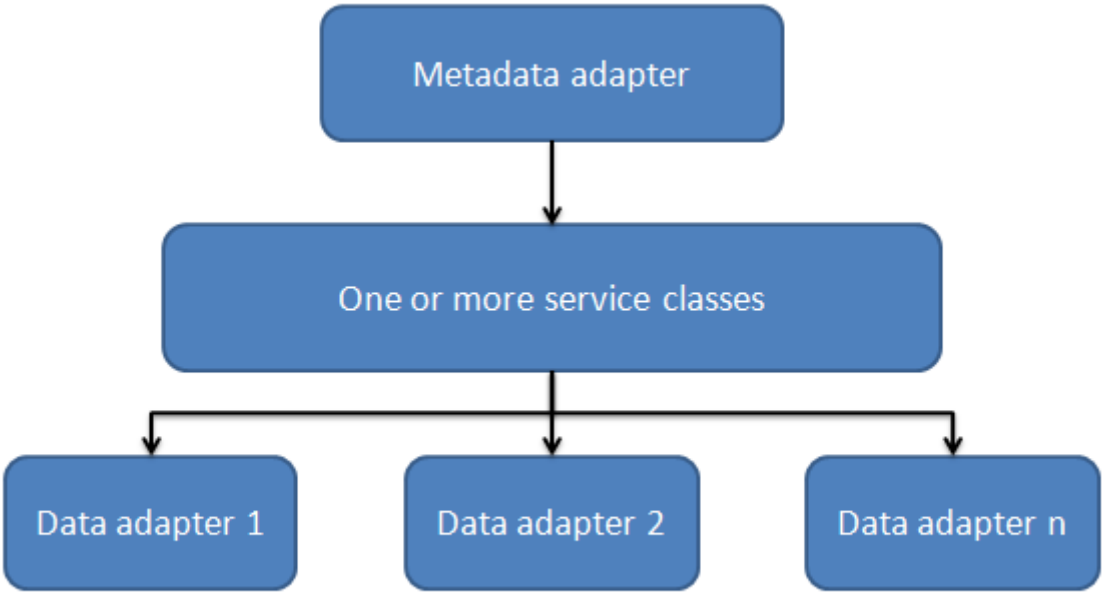
The samples are "documented" through comments in the code, and these are not abundant. Furthermore, the code is rather messy and hard to follow. I had a hard time figuring out what parts was related to Lightstreamer and what regarded the user interface. As a result, I spent many hours debugging both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

Lighstreamer's server comes with a large configuration file where you can tweak it's performance to fit your needs. There isn't too many options, but the file is easy to get lost in. This is because they 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 libraries and frameworks in this thesis, Lightstreamer is huge. It even feels larger than Play and Meteor (TODO: double crossref), 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 a normal application stack. (TODO: figure).

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 Quality of Service (QoS). (TODO: ref general concepts). 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. (TODO: figure).



Lightstreamer uses an application model that resembles a Service Oriented Architechture (SOA). More specifically, it uses a publish/subscribe model. It doesn't follow all SOA principles though (TODO: source). Some examples of this is that it is not discoverable and the strong coupling between clients and 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, needs to have fields that corresponds to the fields in the DOM client side. (TODO: figure).

```
<div class="item">
    <div class="itemHeader">
        <h2><span data-source="lightstreamer" data-field="name"></span></h2>
        <div id="horizontalItemHeaderLine"></div>
    </div>
    <div class="itemContent">
        <div class="itemInfo">
            <label>Itemnumber: </label><span data-source="lightstreamer" data-fi
            <label>Minumum price: </label><span data-source="lightstreamer" data
            <br/>
```

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 driver arhcitecture. There are several drawbacks to this technique though. Serialization and event routing has to be hadled manually. The first is manual no matter (TODO: figure), but converting to JSON addsd another layer of complexity. It also makes you unable to benefit from the different subscribtion modes.

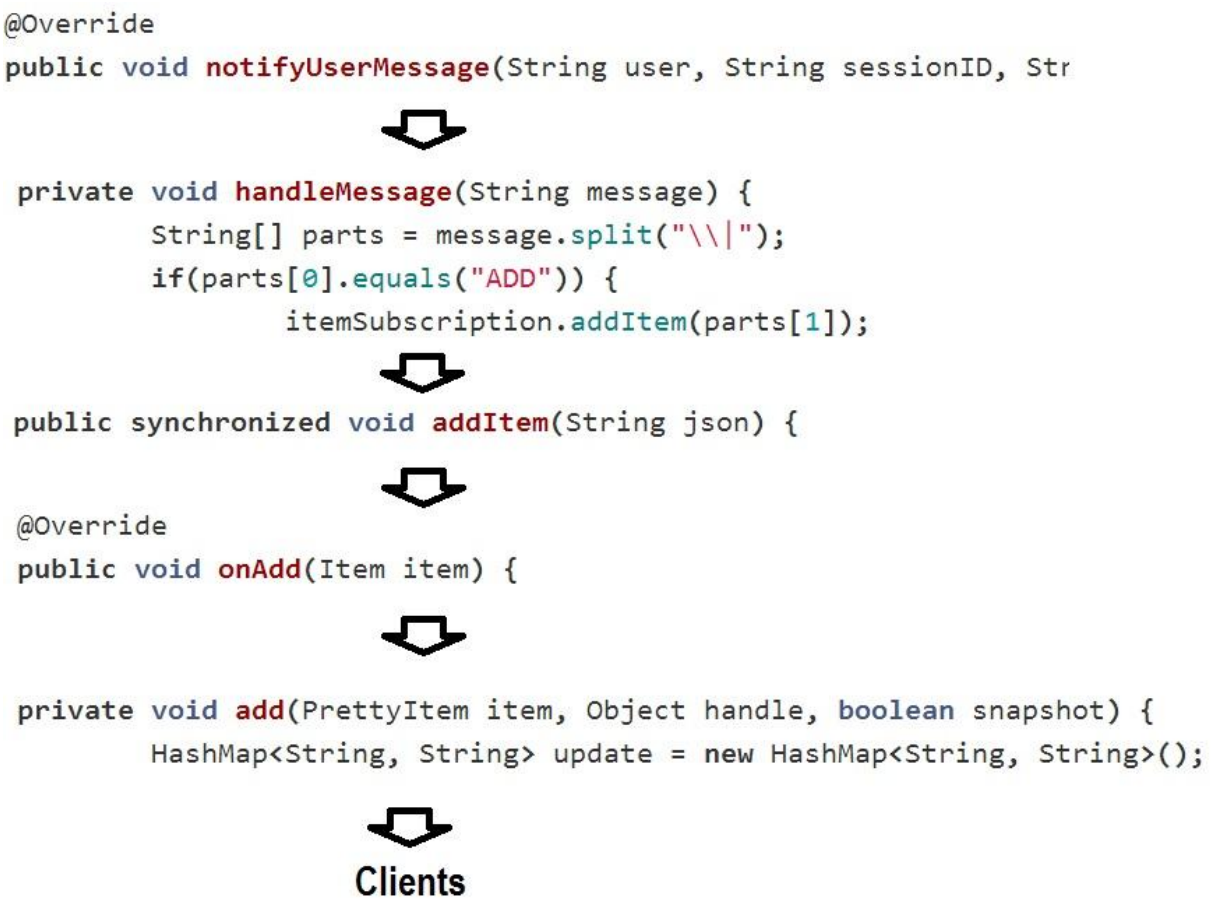
```
private void placeBid(Object handle, Bid bid) {
        HashMap<String, String> update = new HashMap<String, String>();

        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);
}
```

Lightstreamer is not intended to used in this way though, 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. (TODO: crossref).

The tight coupling to the DOM makes using popular MV*[22] frameworks like Anguar or Knockout 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 bewteen the "Lightstreamer-pages" and the rest.

[22] MVVM (Model View ViewModel), MVC (Model View Controller) and MVP (Model View Pattern) are referred to as MV*.

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 figure shows the flow in my simple test application from an incoming message, to a broadcast is sent. (TODO: figure).



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 (in SOA terms). It would be even better if some other entity than the clients functions as producers. This would let the Lightstreamer server do only push, which is what it handles best.

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 is, 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 that 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 dummy data and inject it in the test.

The samples uses an ExecutorService (TODO: javadoc) to handle concurrency. Testing methods that uses this can be a little messy, but no more than testing callbacks of Socket.IO events. As long as you can inject a mock of the listener that the ExecutorService triggers, you are good to go. In other words, the seemingly preffered method for Lightstreamer, is the same I used for Socket.IO. (TODO: crossref).

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 light weight 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 though, is the transport mechanism that uses polling over WebSockets. As far as I can tell, there is no real benefit to this as any browser that supports it, also supports streaming over 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 completletely 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 best 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 uses only Node on the server.

Weswit is actually one of the pioneers for real time technology (TODO: their own ppt). Their main problem now is to keep up with modern day trends. The competition are 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

benefitial. 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. (TODO: partner gardner thingy). 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" exist for a reason. What I think they should do though, is keep everything they got, but also offer what I suggested above.

7.3 Play Framework

I used version 2.1.1, which was the newest at the time. Play comes with a server side template language for rendering of HTML. I minimized the use of this in order to use the common user interface described in section (TODO). Real time communication require JavaScript on the client side. This means that the server side template language is not applicable.

7.3.1 Documentation

Obtaining Play is a simple and well documented process. (TODO: link to installing). 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. These are mostly easy to follow, but I ran into some issues with links to certain files that didn't work anymore.

As Play lets you write in either Java or Scala (TODO: double ref), 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 this 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. Instead of referring you to a change log, Play's documentation lets you browse previous versions. This simplifies the process of migrating to a new version a lot.

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 these itself, but not in too much detail.

Using examples, the documentation is easy to read and learn from. Learn by doing is, in my opinion, the best way. With Play, there was little need for diving into the source code. 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 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 though. Compared to the route configuration of ASP.NET MVC, 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 you page refreshes.
- The template language is Scala. It resembles the template language of other frameworks. Compilation errors from all template files, are also shown as with other files. (TODO: fig)
- Play builds on Akka, which means you have the Actor model available for concurrency handling. (TODO: footer). It is an old, mathematical concept (TODO: source), but it has become increasingly popular recently. As with the rest of Play though, you are not forced to use it.

The real time features of Play are close to bare metal, but it provides some nice abstractions. There is 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. (TODO: figure)

```
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 {
```

With the in-channel you get access to the "onclose" and "onmessage" events of the WebSocket API. "Onconnect" is handled by the "main" WebSocket class. (TODO: footer or figure text). The out-channel handles sending of messages only.

Since HTTP-streaming is from server to client only (TODO: cross?), 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 uses a similar out-channel as the WebSockets class provide. One handy thing the Comet class provides is a callback for disconnects (TODO: figure). 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),
        }
});
```

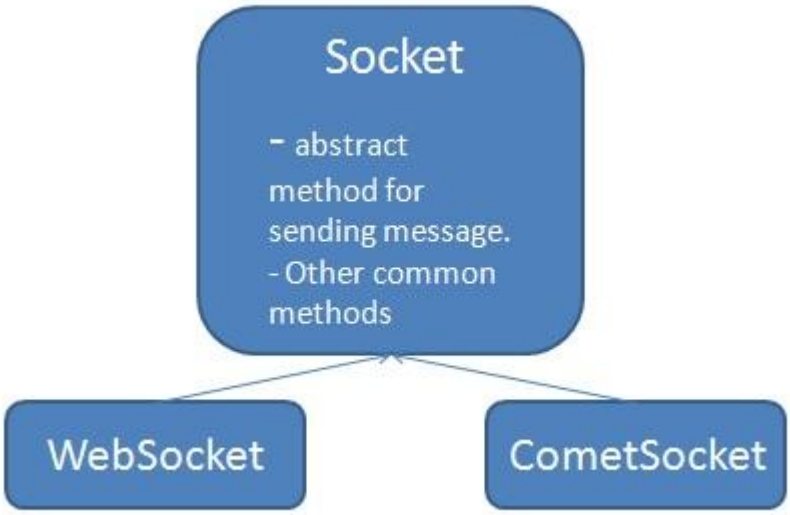
On the client you stand without support. To access WebSockets, the standard WebSockets API is used. Comet depends on the presence of an iframe on the client in order to receive messages. The utility class on the server wraps outgoing data into a function within a HTML script-tag. You specify the name of the function, but it has to be globally available or provided an absolute path[23].

Both of the utility classes allow for sending either strings or Jackson JsonNodes. 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. (TODO: source JavaJsonRequests). There are some drawbacks though, as serializing more complex objects yields more overhead. (TODO: fig) With a lot of nested arrays and objects, results in many loop constructs. Overall, this can lower the performance of the application.

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

```
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);
```

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. 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. The value in the hashmap has to be some kind of entity that contains the means to send 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. (TODO: figure). With these two transports only, this is a simple solution. If I had introduced long-polling as well, it would have been much more complicated.



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. (TODO: example?). This leads to the same issue as the anonymous callbacks of Socket.IO's events. With Socket.IO I proposed a number of solutions (TODO: cross). These can also be provided 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.

All applications needs to 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. I had some issues when testing my application though, as the "Actor system" just shut down. This was a known issue prior to the version I used (TODO: source). Whether you need this feature is another question. Most development environments features a shared development server that tests can run towards.

Play also supplies a simple way to use an in-memory database to test your models. A traditional Java web application using Spring MVC would have a separate data access layer. Play encourages the use of Ebean which puts data access into the models themselves. Testing is similar though, 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. (TODO: source JavaTest). 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[24]. 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 (TODO: example). In retrospect, this was not the best solution. Instead I could have the client find out if it has the WebSocket or MozWebSocket (for Firefox) objects. If it doesn't, it doesn't support WebSockets. This information could then be used to connect to the server. I used this method in the load tests (TODO: crossref).

[24] I tested with Internet Explorer 8 and newer and the newest versions of Chrome, Firefox and Opera.

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

7.3.5 Maturity

The framework was introduced in 2009, but Play as it is today came out in 2012 (2.0 version, TODO: philosophy). With the 2.0 version, 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. Although it's popularity is almost increasing by the minute, it still isn't used that much. The TIOBE Programming Community Index , places Scala as number 37 in august 2013 (TODO: source + update?).

Nonetheless, Play keeps gaining popularity, and it is considered a stable piece of software. This is my opinion as well, but there were some issues. Outdated versions of third party software and broken links on the homepage (TODO: crossref) was the most prominent.

There is no arguing that Play follows modern trends at least. 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

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 to host the application. An ASP.NET MVC 4 application wraps the SignalR components. No other communication than serving static files uses the MVC specific aspects of the application. The application uses the common user interface described in section (TODO).

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

7.4.1 Documentation

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

During early development, the only documentation was up on GitHub. (TODO: docs on git). Even so, there were 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 reside in one place. (TODO: source docs).

The new documentation has a lot of tutorials and examples. It is way more comprehensive than what it was on GitHub. Just as Play does, SignalR's documentation lets you browse previous versions, though not with the same granularity[25]. 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[26].

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 have to use this or change it to something else. Still, the term "IOC" isn't even mentioned in most of the other frameworks' and libraries' documentations.

There are some things I missed in the documentation. Hubs are thoroughly , but Persistent Connections are not covered in detail. The GitHub pages covers this though, so it may be in the making for the new pages. Furthermore, I missed some documentation regarding testing hubs. Finding information of this required me to search elsewhere (TODO: source).

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. I have focused on hubs, as I predict this to be the most common usage of the library. Persistent Connections are a lower level of abstraction than Hubs. It resembles the WebSockets API, but with some extra methods for handling sending, broadcasting, 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.

---

[25] 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.
[26] Class references in C# are similar to Javadocs for Java.

The Hub API strives to be very simple, providing a RPC (Remote Procedure Call) model which makes the code simple and understandable. A Hub exposes all its public methods to the client and they can be "called" directly. (TODO: example).

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);
}
```

C#'s convention is to have method names with capital first letter, while JavaScript has the opposite. As a result, the name of the method or function you "call", doesn't match the actual name[27]. It takes a little while to get used to, but in the end it makes sense to follow conventions.

Hubs manages 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 out to the clients. Either via 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 get call. Clients can also specify functions that are callable from the server in the same RPC style as clients call server methods. (TODO: figure). 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 (TODO: cross).

[27] Following conventions you would ”call” a function called “someFunc” on the client. The server has named this “SomeFunc” though in order to follow C# naming convention.

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);
```

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 (reduce bandwith). To get this back in readable form on the client though, you have to write some manual code (TODO: ref performance). The serialization even handles date objects. I ran into a problem with this though, as dates 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 normal web application, just as Socket.IO. This means that the only entities you have to test are the Hubs or the Persisntent 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 though, is not feasible. This is because you have to mock the "Clients" object as this has dependencies you don't wan't 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. (TODO: ref unit testing hubs). Setting up tests like this allows you to verify all logic within a hub. (TODO: example).

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

The site also claims that you can test that the Hub sends out correct values as well. I was not succesful in doing this though. 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, TODO: crossref) as transport in a browser that supports Server Sent Events, you’re not allowed.

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. (TODO: ref releases git). 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. (TODO: vimeo video). 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 way to get support for WebSockets. This isn't due to the implementation of SignalR though. (TODO: crossref).

7.5 Meteor

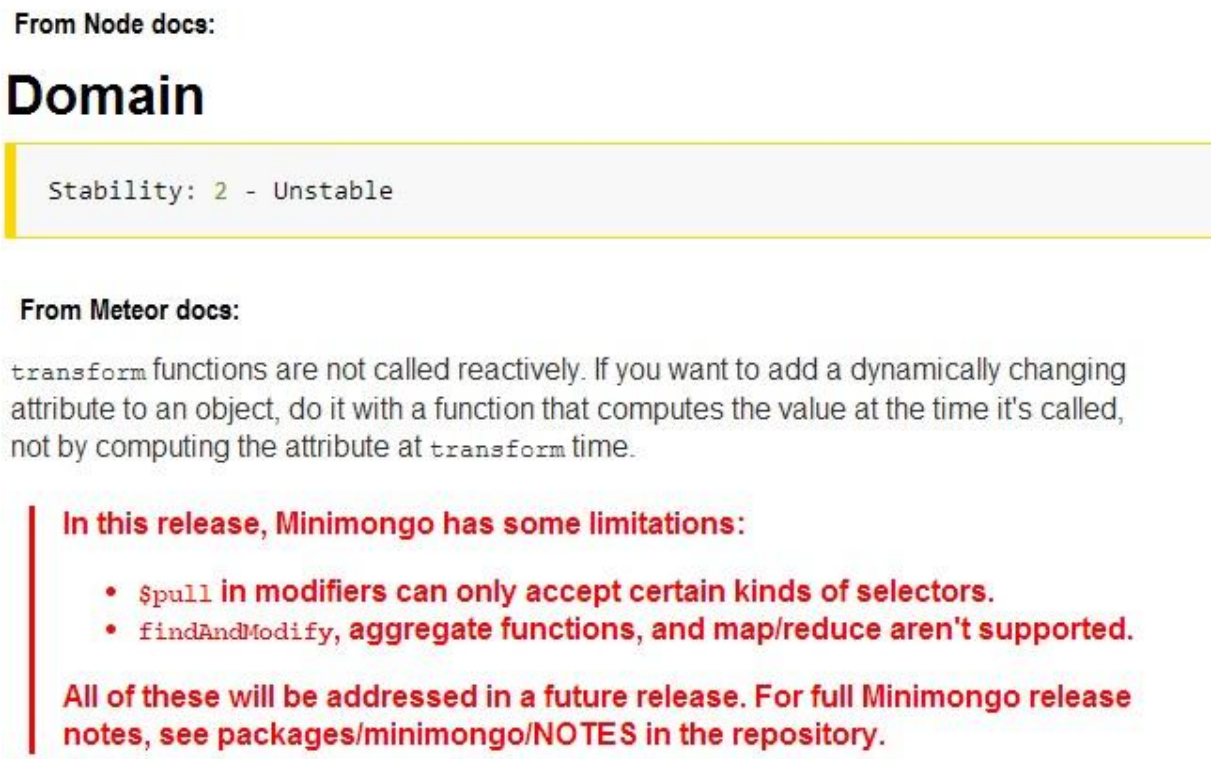
The unofficial Windows version is a little behind the current version of the official version. During my work, this was not the case, and both versions was 0.6.6.3. Meteor did not have support for MySql at this time. I used the supported MongoDB instead. This means that I was not able to use the common functional test case described in section (TODO). Meteor also have integrated support for a client side template language. The common user interface described in section (TODO) 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 though, since you have to go via the "supported platforms" site. (TODO: supported). 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. (TODO: Daily JS). 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 is 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. (TODO: figure).



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. This functionality breaks sometimes though. 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 require 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[28].

7.5.2 Simplicity

Simplifying developing web applications is the motivation behind Meteor. Right now though, it is not simple. But I can see that if the smart package system (TODO: docs) 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.

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 the 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. (TODO: figure). I must say that I prefer to handle this my self, as I think that it makes the code more readable.

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     }
```

Meteor encourages the declaration of global level[29] 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 (TODO: docs). Any front-end developer would have second thoughts on this practice. General JavaScript development discourages the use of global variables. (TODO: source needed?).

Another thing that is a little tricky because of Meteor's "magic" is structuring files that depend on other files. Files load in a specific order, based on how deep they are in the tree 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.

---

[28] You can see one of the videos here: https://www.meteor.com/screencast
[29] If you declare a variable without the keyword "var" in front of it in JavaScript, you declare it globally.

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 (TODO: cross), 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. (TODO: example). 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");
                }
        },
```

Having direct access to the database from both client and server is something 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. (TODO: screencast). From a users 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". (TODO: roadmap). 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 though, they feel a little unnatural, like your hacking the framework to get it working. My approach was

using Node with Mocha as test runner. Then I used a module that allows you to inject mocks into another module. (TODO: unittestling and example).

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

Meteor files are not Node modules though. 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. (TODO: another example).

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

Testing Meteor is something that will best be done by integration testing of some form. Laika (TODO: source) is a testing framework for Meteor that does just this. 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[30], something they wouldn't if it didn't look promising.

[30] 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.

7.5.4 Browser support

Meteor supports all major browsers. I had some issues regarding the WebSocket support of SockJS though. 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, and pointing the browser to 127.0.0.1 solved the problem (TODO: source).

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 (TODO: source). This assures that the team can work on Meteor full time rather than besides other work. (TODO: meteor/about/people).

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 applications. A repercussion of this though, 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 revamp web applications. Do they promise too much? Only time will tell, but if they deliver, 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 APIs 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 1.0. 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.

Project part 2:

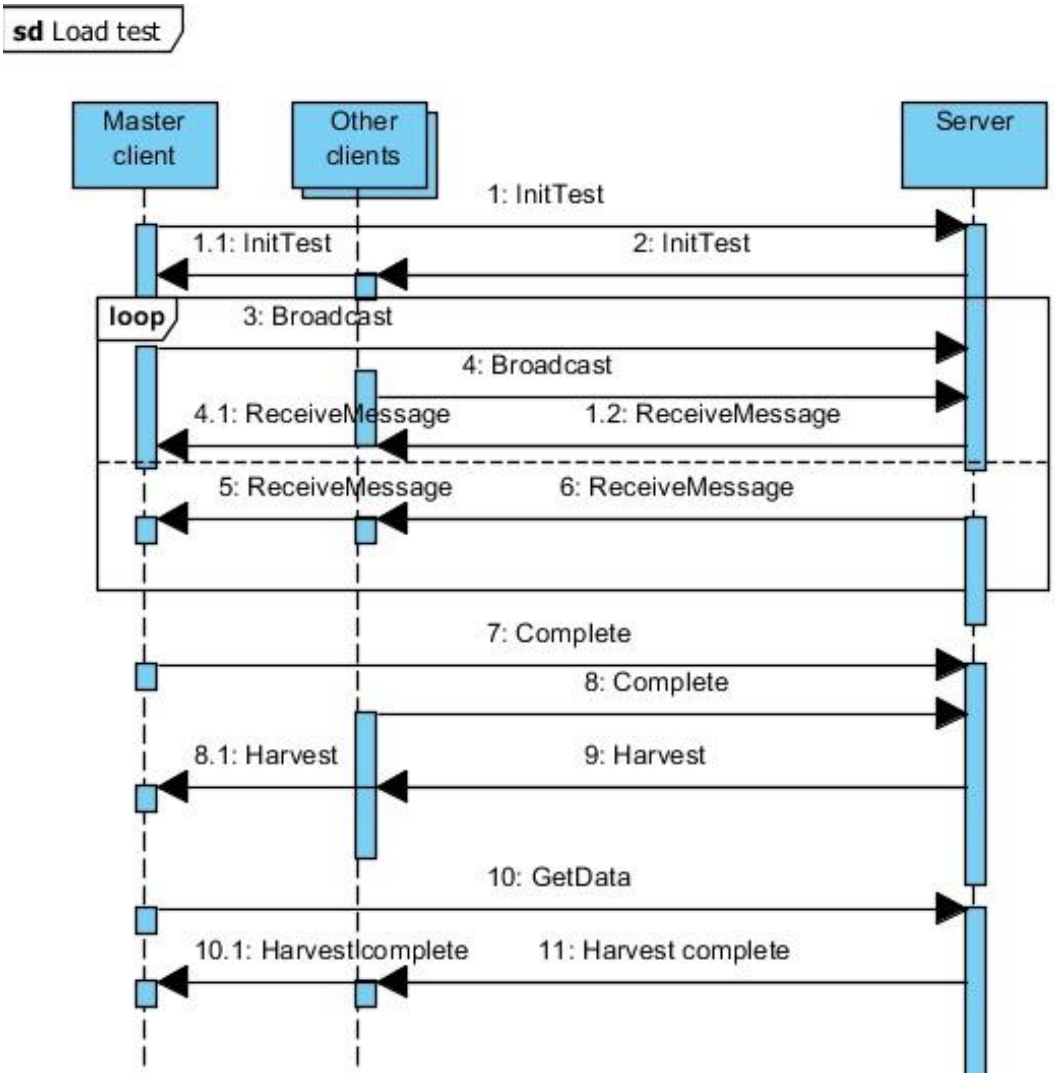
Load testing

8 Methodology

This part will focus on performance. (TODO: crossref problem statement). Both for the individual frameworks and the different protocols. I will strive to answer questions about 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 have designed follows this concept. A set number of clients will follow a strict message flow, recording data along the duration of the test. One of the clients will be dubbed "master" and handles starting the test. Figure (TODO) describes the flow of a test. Section (TODO) provides further insight into this.



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.

- 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 generated graphs. Numbers on the axis represents time intervals in seconds. For instance, the value “0”, represents the interval from zero to one second. I ended up with running short tests, so this value was always set to one.
- The start time as recorded client side. To eliminate time differences, the server will record the start time as well. Calculations for client data will use the first, while server data will use the latter.

All messages from the "complete" message handle exchange of data. At the end of a test, clients will have 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.

8.2 Test data

While the tests run, both the clients and the server will 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 in a given interval. It also registers how many messages it received and sent in a given time 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 will be registered immediately after receiving a message. The registration of messages sent from the server, will be registered as close to the send event itself as possible.

The other telemetry measured automatically by the tests is latency data. Each client need to calculate the latency of its own messages. To achieve this, a message will be given an unique id. A client then registers that it has received a message only once, and calculates the latency using the timestamp for when it was sent. It is one drawback to this technique though. 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 above, I will monitor some telemetry with other tools. What a certain framework or transport require of the server in terms of memory and processor will be measured. Network usage is the data I expect to see the clearest differences between transports, along with latency. Hence, it is important to measure how many bytes gets 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[31].
- Using real browsers.

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

Most load test setups I have seen uses some sort of desktop or console application to emulate clients. (TODO: give examples? Crank, Netling...). This test however, has many aspects that I have yet to see in any other scenario. I am going to test five different frameworks. All will use WebSockets, but I will also test 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 they are separate protocols. 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 provide. 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. Phantom is the most widespread, but the current version (1.9) does not support WebSockets (TODO: source). 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.

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 provide. 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 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. (TODO: git). Firefox turned out to be the ideal browser as Selenium has good support for it. It was also the browser that handled several clients best.

---

[31] A headless browser is a browser without the graphical user interface (TODO: what is a headless browser - blog).

I used only 60 clients running in 30 browser instances. Message frequency was set to two messages per second per client. The load this produces is not substantial, but it enough to generate differences between both frameworks and transports.

I used my own Asus K55V laptop to host the clients. 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.

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 common to all frameworks. 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.

Each of the frameworks need different servers, but I wanted them to follow a strict implementation. There are two main entities for registering data: the "loadhub"[32] 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 Framework 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 (TODO: see raw data). Therefore, I decided that 10 runs of each combination of framework and transport would suffice.

8.6 Displaying data

I chose to use Highcharts (TODO: link) to display charts with all collected data. This is a simple and powerful API that provide graphs that suit my need. To format the data gathered by the clients and servers, I designed a simple Web API. After a test is done, the master client is in possession of all the raw data. This is then sent to the Web API. The API then processes the data and returns three objects that the

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

Highcharts API can make use of. A total of 33 tests ensures that also this part of the application stack performs as expected.

While the Web API provide charts for each run, it is the average values of several runs I need. To help with this, I designed a "Chart merger". This contain a series of (unit tested) functions to extract the average values of different types of tests. It then presents this in Highcharts graphs. These graphs will be used in the thesis. The chart merger also handle the data I gather manually.

8.7 Configurations

With SignalR, IIS benefits from a few configurations. I followed the steps given by the wiki of SignalR's GitHub page. (TODO: source).

Lightstreamer required some more work before it was ready for testing. First of all, I had to upgrade to the best edition available, the Vivace edition. Weswit proved very helpful, giving me free access to this version for the duration of my work. They also helped with a few settings needed for the server.

My initial results showed that Lightstreamer lagged behind all the other frameworks, so I turned to Weswit for help. Appendix (TODO) shows the whole mail correspondence with an engineer at their office. 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 long running test showed that the run stabilized around this mark. Hence, there was no need for a longer run.

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 me 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 (TODO: page), Wireshark (TODO: page) and Fiddler (TODO: page) were the most prominent that I considered. In the end, the choice fell upon Wireshark as it provided exactly what I needed.

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 (TODO: source). 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 percent a process uses. Because of this, the server would always show close to

100% usage[33]. 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 the V8 engine it builds on, is implemented in C++. C++ compiles to machine code, and it does not need a runtime to function. Both Java and C# code compiles to byte code that has to be interpreted by a runtime. For Java this is the Java Virtual Machine (JVM) and C# has the Common Language Runtime (CLR). As a result, C++ is a faster language than both C# and Java (TODO: source).

I chose a coarse grained method for monitoring resource usage of the different frameworks. As a result, the results for processor usage 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. I will perform an analysis of this results in chapter (TODO).

No matter what method I use to measure machine resources, I still get an overview of how each transport perform. This comparison cannot be 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.

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.

---

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

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 could 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 specification states that no client should have more than two connections to a single host. (TODO: source).

Nonetheless, most modern browsers have increased this limit to somewhere between four and eight. 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. A browser takes up about 100 megabytes of RAM when idle. Considering that each browser will store some data as well, this number will grow 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 server to client 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 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. 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 calculation of the theoretical throughput of each test kind. The basis for each calculation is a capture with one client sending 10 broadcast messages in two seconds. For these captures, I started the capture before I navigated to the test page, thus ensuring that all packets got captured.

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. I must thus 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 though, that there may be some differences between how browsers handle different transport mechanisms. Still, it gives an indication.

The basis for the calculations is on GitHub (TODO: link and maybe explain a little? Or make readme..).

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. I was able to get a basis for calculations of network traffic for it though.

8.14 Testing idle connections resource usage

To find out whether WebSockets has potential to set a new standard for client/server communication, I will measure 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 will perform two tests with each framework to test this. One will connect 1000 clients evenly spread across 10 browsers to the server. The other will connect 4500 clients. This will be done using 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. After this, some connections were refused. With a safety margin on 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, I wanted to add a safety margin here as well. To be safe, I landed on 25 browsers.

8.15 Raw data

All raw data can be viewed on GitHub: (TODO: link and try to put captures there).

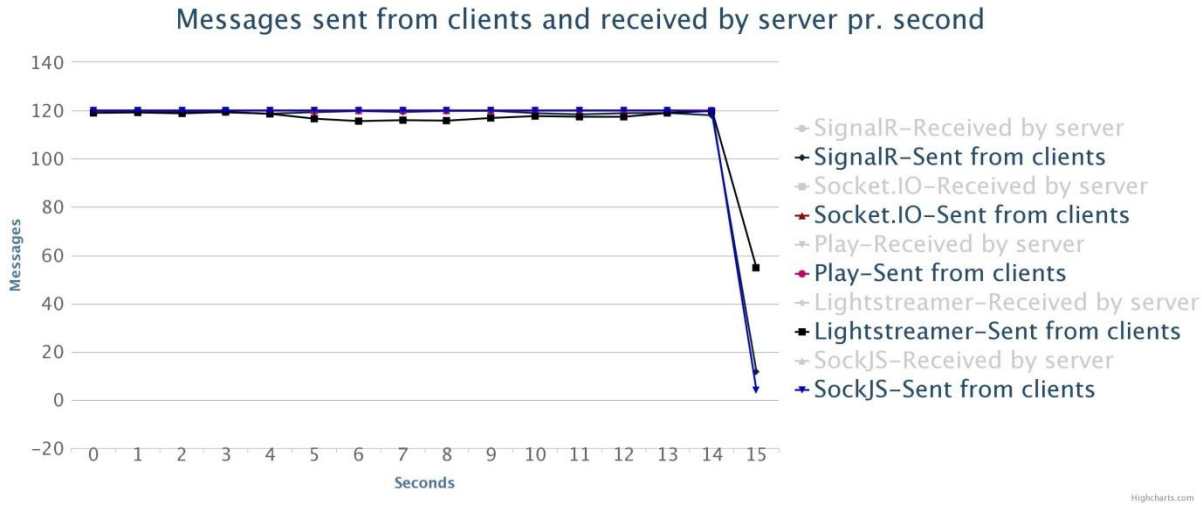
9 Results

This chapter describes the results of the tests. Each graph show the average values of the 10 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

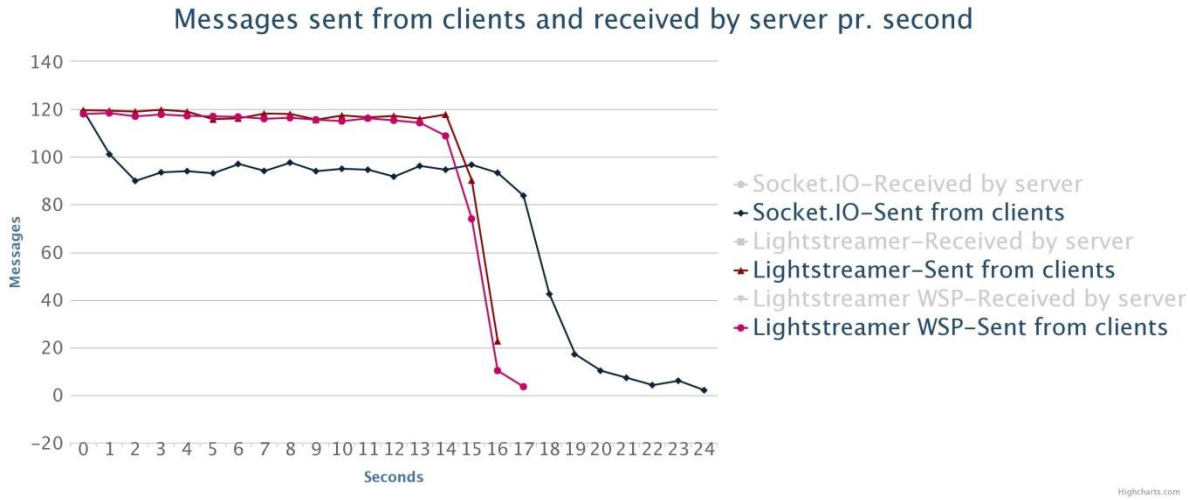
general trends in the results as well as anomalies I experienced. All graphs can be reviewed in appendix (TODO). An analysis of the results is given in the next chapter.

9.1 Messages sent from clients

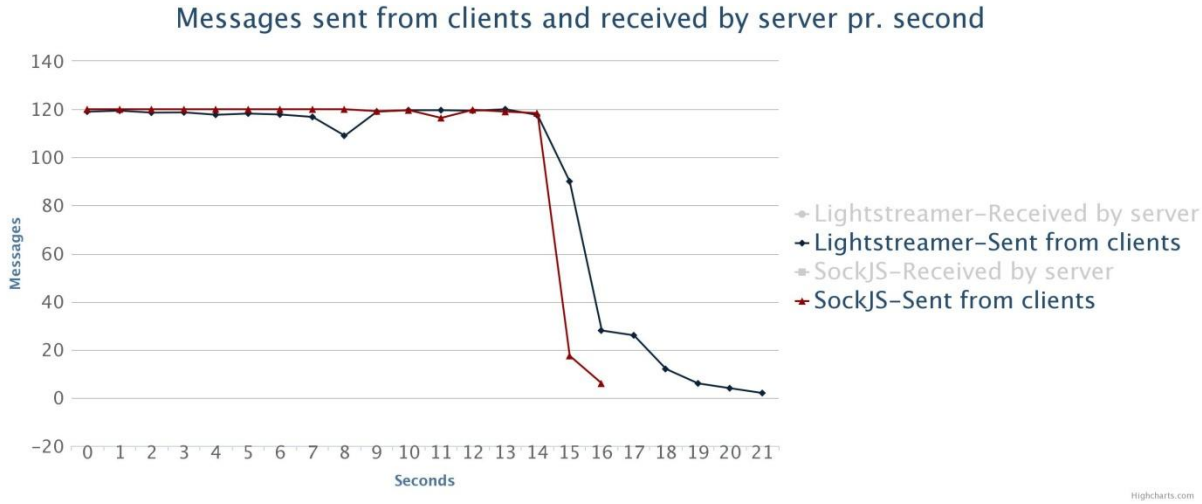
With 60 clients and two messages pr. second, I expected a stable graph with 120 from 0 to 14 for a perfect run. A lot of the runs show this behavior, but a lot more than expected have certain anomalies. Figure (TODO) shows the results for WebSockets. It displays one example of such anomalies. Some runs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



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



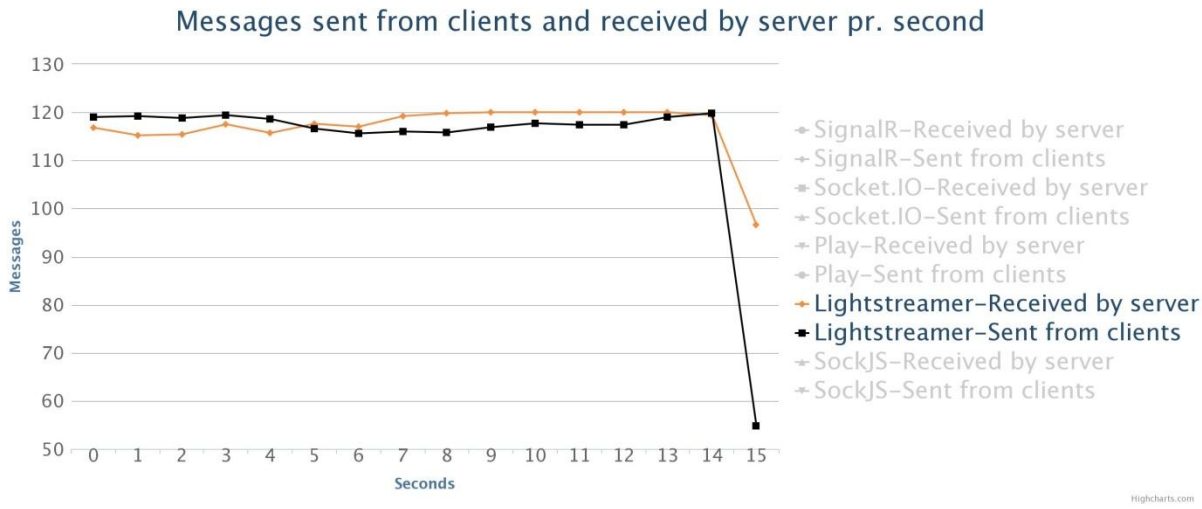
The results from HTTP-streaming show a lot of overtime for Lightstreamer. (TODO: figure). This reflects only one run that lasted seven seconds too long. None of the other runs replicated this effect, but they still ran one or two seconds of overtime.



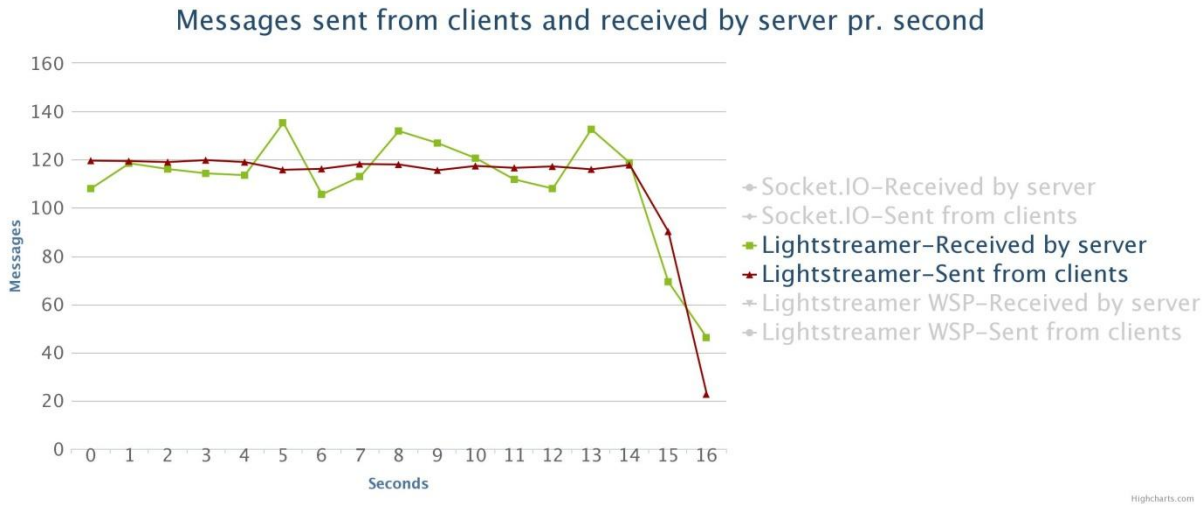
9.2 Messages received by server

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 (TODO), 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.

Again though, there are some anomalies. Figure (TODO) 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 it is not a perfect run. I will discuss possible reasons for such variations in the analysis chapter. (TODO: crossref).



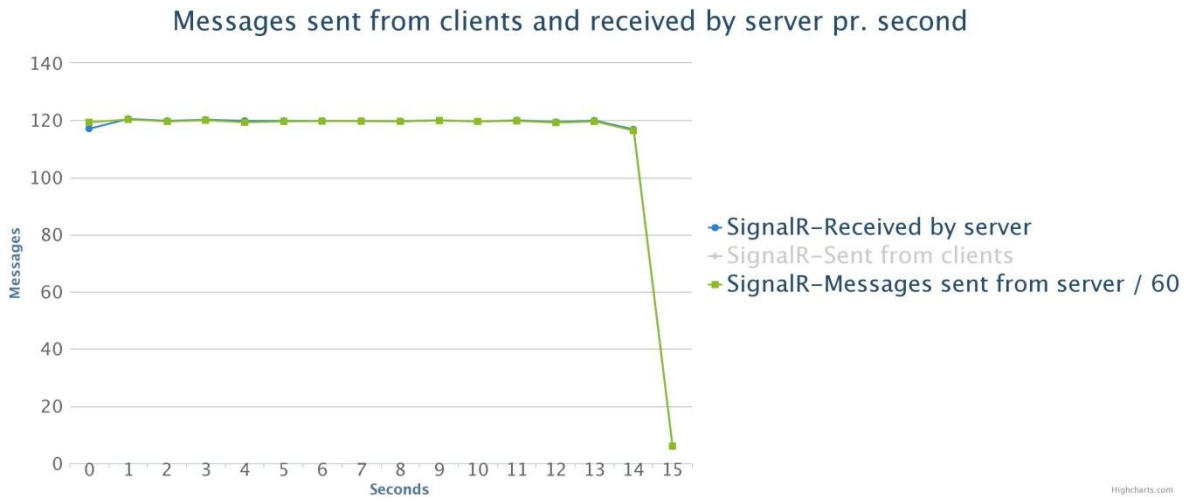
While most of the variations are small, polling is, once more, the largest source of anomalies. The clearest example of this is Lightstreamer. (TODO: figure). It seems like the server worked in intervals, with peaks of more than 130 and bottoms of 105 received messages. These are variations of up to 12,5%[34], 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 (TODO) show 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.

[34] A variation of 15. 15 / 120 = 12,5%.

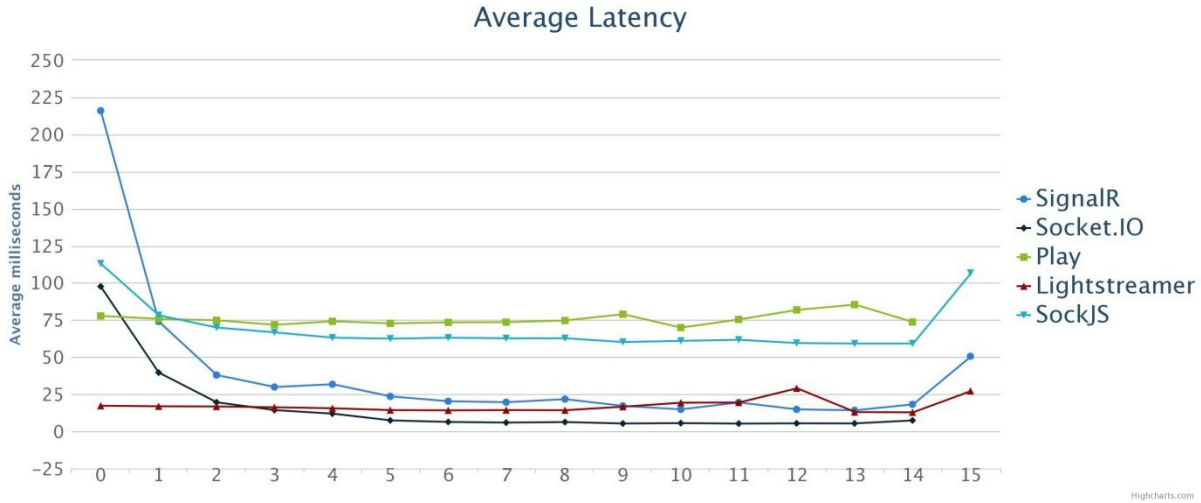


9.4 Average latency

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 (TODO: crossref).

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

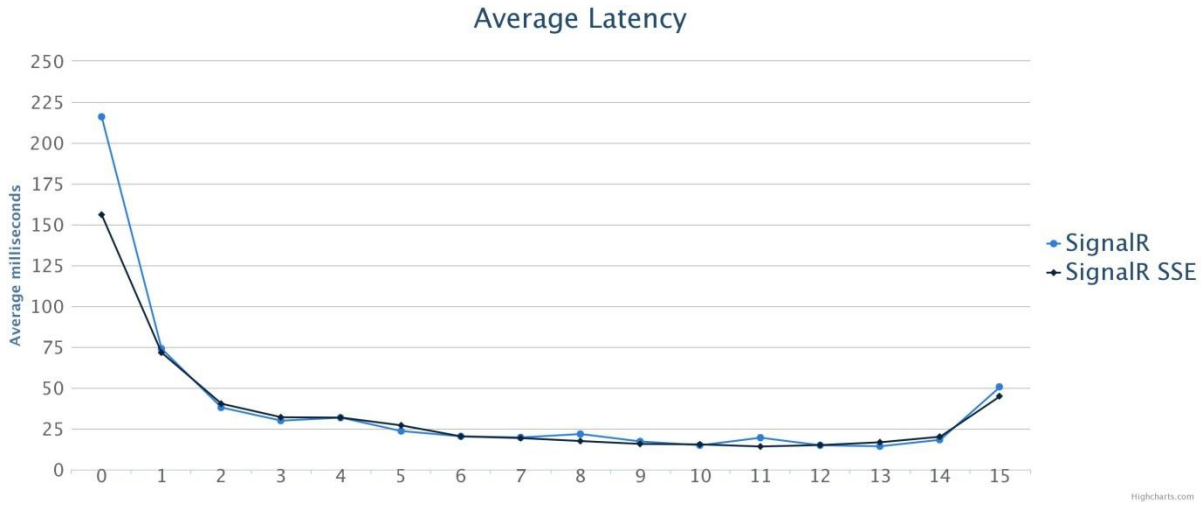
9.4.1 WebSockets



Both Play and SockJS relies on some manual client handling. Broadcasting messages with these frameworks relies on a loop construct. The other three have internal mechanisms that handles this. Figure (TODO) shows a clear divide between the first two and the other three. Of the three best, 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.

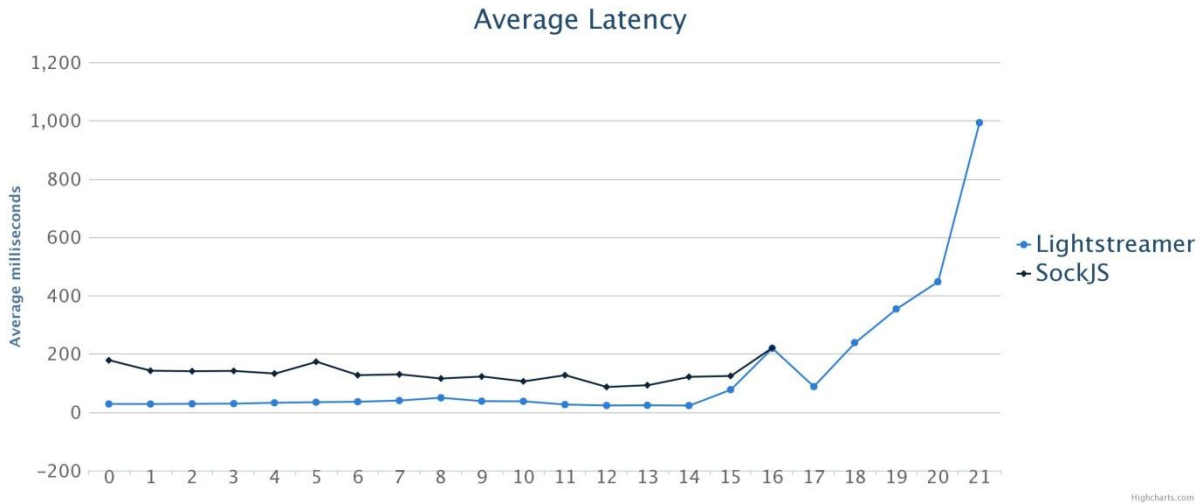
9.4.2 Server Sent Events

Only SignalR provide Server Sent Events as one of the transports. The graph below (TODO: figure) compares the results of this transport to SignalR's results using WebSockets. This is a very interesting result as it shows that Server Sent Events matches WebSockets when it comes to latency.



9.4.3 Http-Streaming

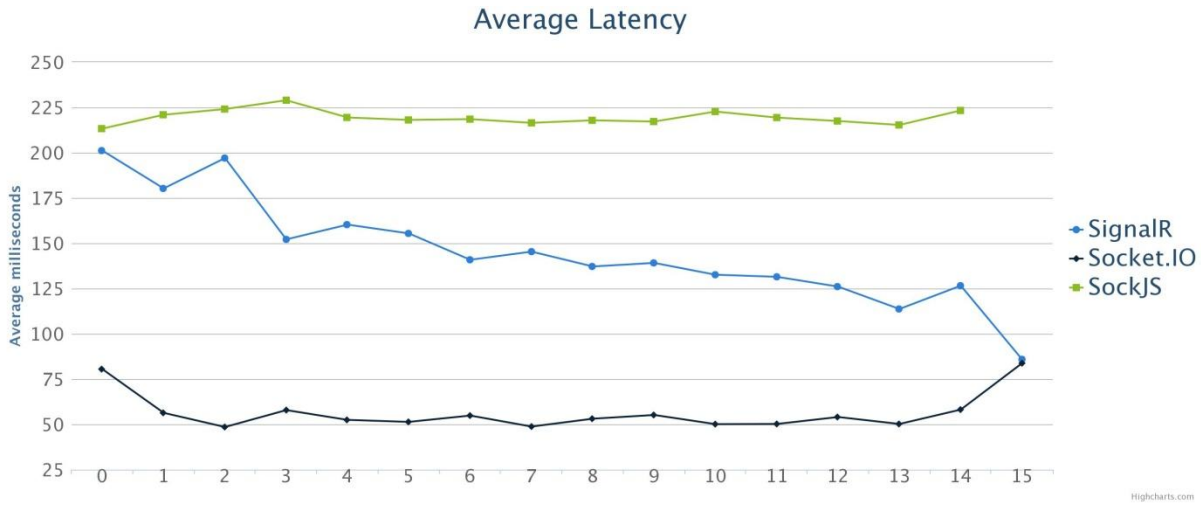
In section (TODO: sent from clients), I mentioned that only one run with Lightstreamer had seven seconds of overtime. The graph shown in figure (TODO) reflects this fact. As the other runs never passed two seconds of overtime, the data from 18 to 21 seconds should be disregarded. Prior to the 18 second mark, Lightstreamer performs well with HTTP-streaming. The latency is higher than WebSockets, but not much. SockJS does not come close to the same. Here though, the difference to WebSockets is more clear than with Lightstreamer.



9.4.4 Long-Polling

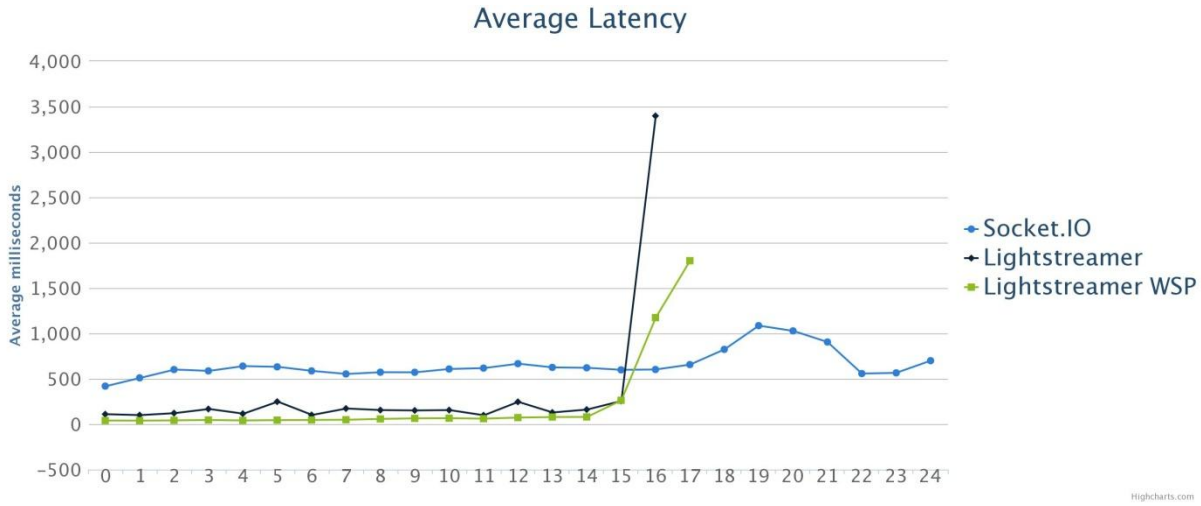
As with WebSockets, Socket.IO outperforms the other frameworks with long polling. However, as the graph shows (TODO: fig), SignalR's latency decreases throughout the

duration of the test. As discussed in the introduction to this section (TODO: cross 9.4), 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 around the ten seconds mark.

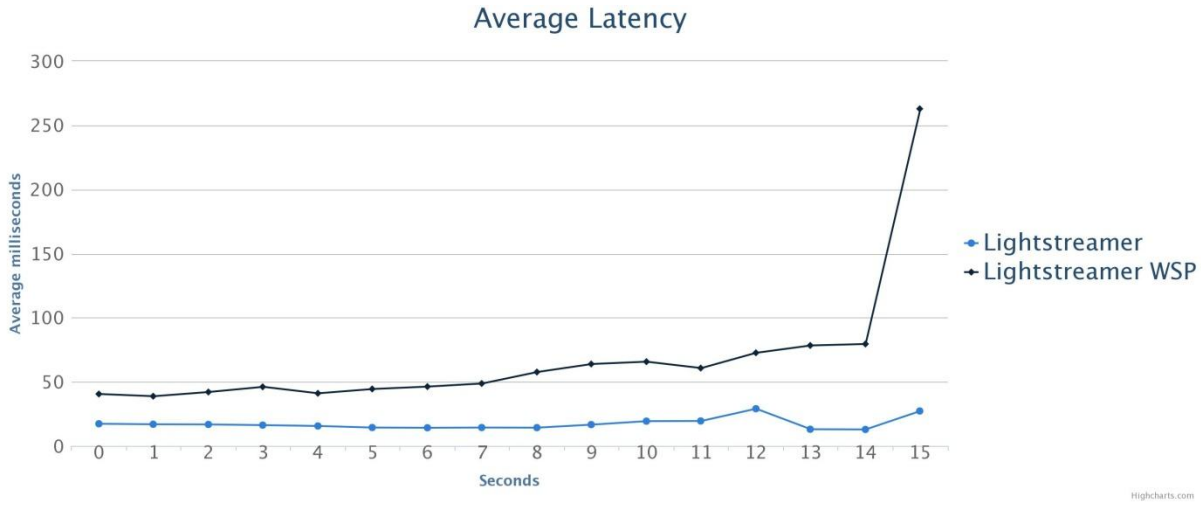


9.4.5 *Polling*

Once again, polling is the greatest source of deviating results. As the graph (TODO) 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.

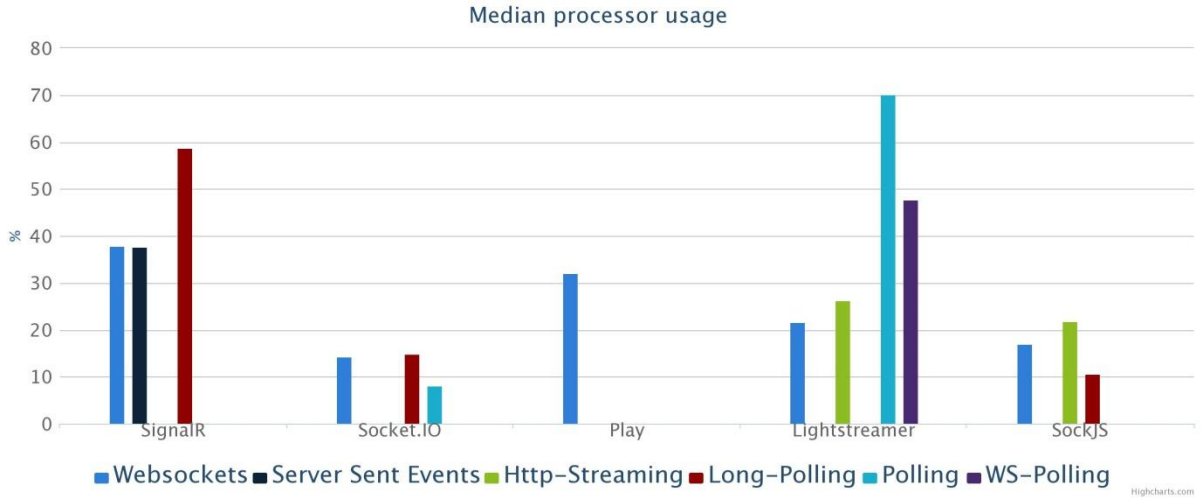


Polling using WebSockets is more efficient than over HTTP by about 100 milliseconds. This is not surprising, but I still don't see the use case for this technique. As a browser has to support WebSockets to use it, it should be compared to streaming using WebSockets. Figure (TODO) compares the two, and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



9.5 Median processor usage

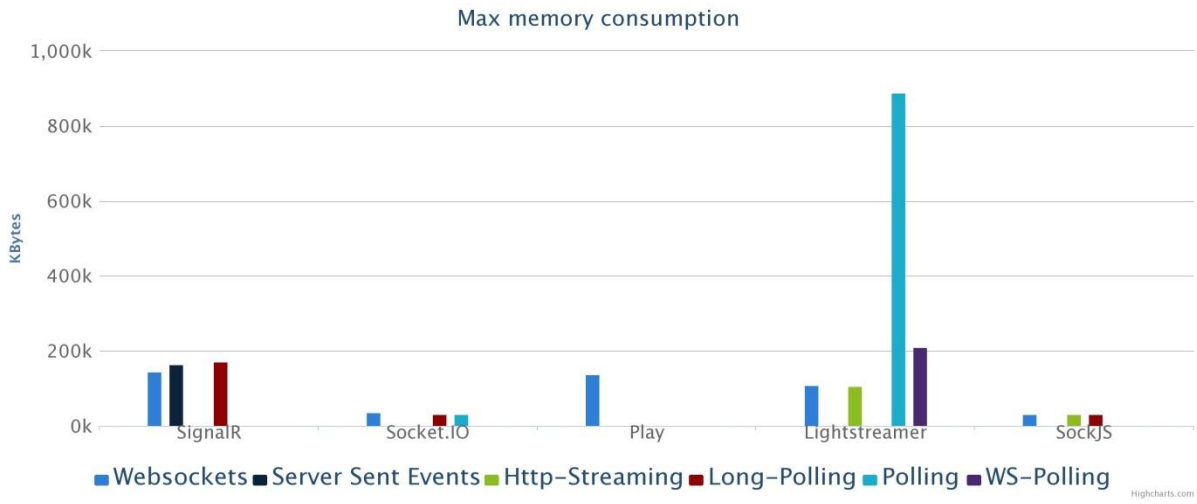
As discussed in the methodology (TODO: cross), these result reflect the usage of the servers, not just the frameworks. The lightweight nature of Node shines through as both Socket.IO and SockJS takes up about half as much as any other (TODO: fig). Otherwise, it seems that the Java solutions perform a little better than SignalR. The analysis gives insight into why this is so. (TODO: cross)



9.6 Maximum memory usage

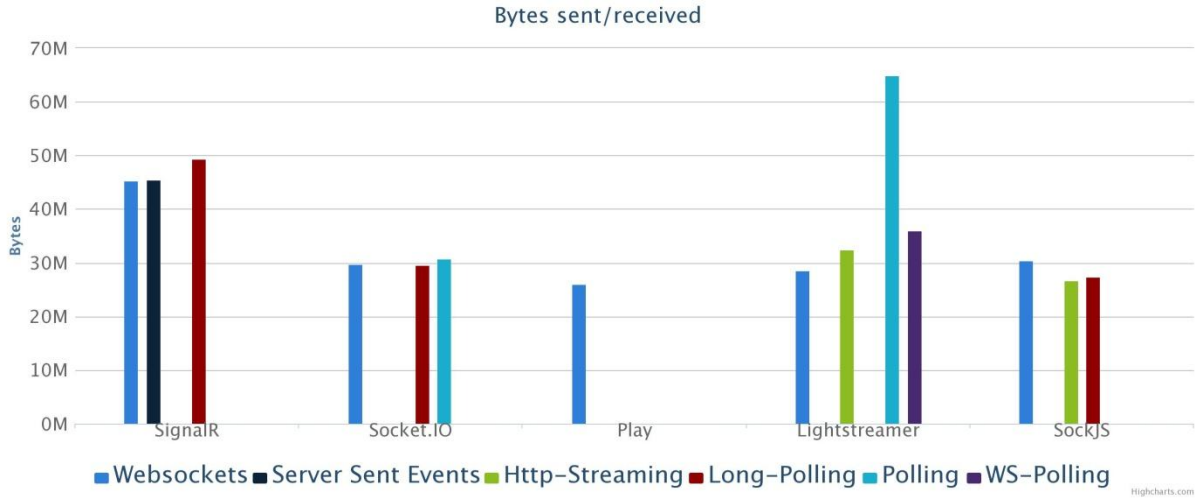
Again, the results reflect the resource usage of the servers as well as the frameworks. As with processor usage, one can see that Node outperforms both Java and C#. (TODO: figure). Otherwise, we see that SignalR, Socket.IO and SockJS have almost the same memory usage regardless of transport. Polling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

There is one source of error in the results though. I measured the memory consumption at the end of the test. While I did not see any drops during the tests, it was some drops that occurred just after the test finished. Some of the raw data stands out from the rest because of this.

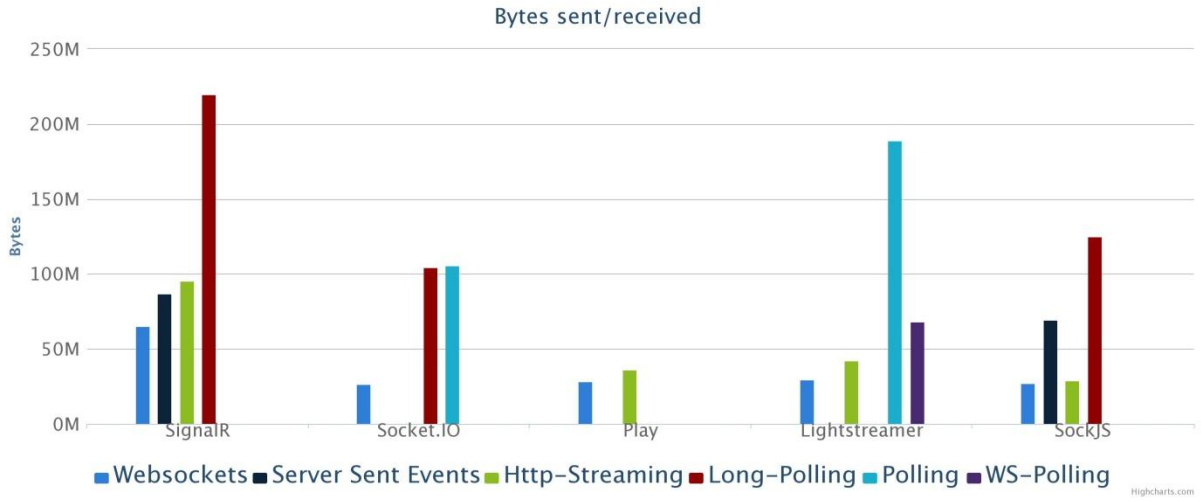


9.7 Bytes sent/received

In section (TODO), 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 is a result of the recorded network traffic. (TODO: fig).



The next graph is the data I calculated. (TODO: figure). It shows that SignalR sends a lot more data than any of the others. 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.



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 (TODO) shows how much memory each framework used with 1000 and 4500 clients. Processor usage was mostly 0 for all. Deviations are due to small peaks that each framework had at set intervals. The true processor usage was therefore nothing for all frameworks regardless of how many clients were connected.

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

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 will discuss the results of the tests. I will try to provide plausible explanations to some of 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 discuss the results from sections (TODO, TODO and TODO). 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 from the raw data. The tables also show an average value. This is calculated based on the values in the corresponding graph.

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, 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. (TODO: cross ref method). The clients starttime was set before the call to the "initTest" procedure. (TODO: see section 1 of method). 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.

The table below shows data for messages sent by the clients (TODO). Most cases show an average number that is smaller than the expected 120. All these are the result of tests that went into overtime. A lower number, represents more overtime. The case with the runs that only used one second extra, is that they often registered less than 10 messages during this second. Low numbers for SignalR and Socket.IO (except for polling) is due to this effect.

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

While Socket.IO shows a highest number of 120 for polling, its average is a lot lower. As you see from figure (TODO: cross results), 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. This may have blocked other incoming requests for a short period of time. As a result, the clients can have reached six outgoing requests which is Firefox's maximum.

(TODO: source). Together, these phenomena result in a queue of request forming at both the clients and the server. Then, 10 seconds of overtime isn't too improbable.

The clients 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. I ran tests with all frameworks to see the resource usage with many idle clients using WebSockets. (TODO: cross, put in results!). With Lightstreamer, I was not able to connect more than about 2500 clients before the browsers used all my computers 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 with a cached thread pool to handle concurrency. (TODO: source javadoc). 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 what so ever. I have found indications that a cached thread pool used in conjunction with synchronized code, degrades performance. (TODO: source). This is exactly how I registered data in the Lightstreamer application. (TODO: example). If this is the case, more than just the frequency data may have been affected.

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

executor.execute(task);
```

The table (TODO) shows data for messages received by the server. There are some isolated cases of more than 120 messages received for several 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 (TODO: 9.4), I mentioned that SignalR could have benefitted from a warm up of the server. This behavior seems to support this.

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

With HTTP-streaming though, 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 was sent rather late. It is not possible to tell for sure, since the capture say 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 can have caused the clients to reach the connection limit.

The final table regarding message frequency shows messages sent from the server (TODO). This data correspond 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.

		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 (6001,41)

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 received. Something happened on the server that caused it to skip a whole second. Immediately after, it sends the "skipped" messages, resulting in the high peaks you see in the tables (as much as 15060 sent messages with polling). Network captures from these runs show that it is not because of 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 discuss the latency results. The results (TODO: cross) 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. There are two main reasons for this: Some frameworks had anomalies in the first couple of seconds, indicating that they could have used warm up. Others had anomalies towards the end with a lot of over time. 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. (TODO: table) 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.

	Polling	Long polling	Streaming	Server Sent Events	WebSockets	WS-polling

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

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[35]. Moving from long polling to polling shows a factor of 11,51[36] 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 the double amount of packages. This indicates that Socket.IO bundles more messages into single responses than what Lightstreamer does. Doing this probably cost time, resulting in the higher latency.

Socket.IO in general performs a lot better than even Lightstreamer and SignalR in the tests. With a higher load this would not be the case. A single threaded server will 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, though.

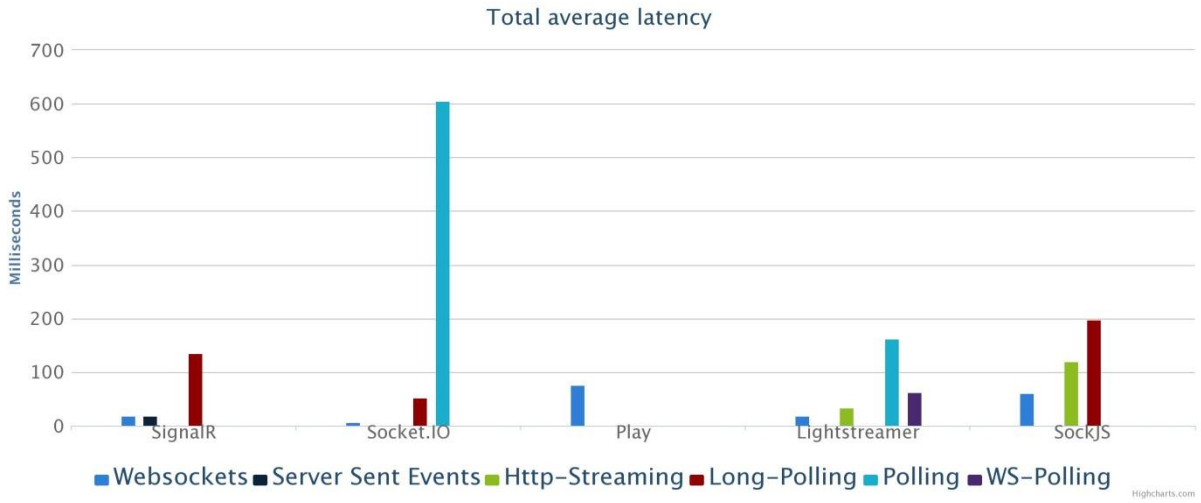
Both Play and SockJS are more manual approaches. SockJS handle many real time features, but broadcasting is not something it supports out of the box. These frameworks relied on a loop to send messages to all clients, a solution that is obviously far from optimal. As a result, they are far behind the 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 long polling beats SockJS's WebSockets.

10.2.2 Transports effect on latency

The graph in figure (TODO) shows the average values from table (TODO). Across all frameworks, WebSockets is the transport that performs best. The only exception is

[35] 161,7 / 33,1 = 4,89.
[36] 605,6 / 52,6 = 11,51

Server Sent Events that perform just as well[37]. HTTP-streaming isn't far behind, but here there is a clear increase in latency. Long polling and polling are, without doubt, a lot slower than any streaming technique - WebSockets or not.



The low latency of HTTP-streaming surprised me a little. It has an average latency of about twice as much as WebSockets. 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. It is clear to me that HTTP-streaming is a reliable technique to build pure push applications upon.

Considering that Server Sent Events also rely on HTTP-requests to get messages from a client, its performance is impressive. Higher loads may benefit WebSockets over both techniques. A blog post by William P. Riley-Land suggests that WebSockets perform better with higher loads, but not much. (TODO: source). This test used two different libraries for the different transports: connect-sse for Server Sent Events and Socket.IO for WebSockets. (TODO: sources).

Riley-Land claims that the results are "in the same order of magnitude" (TODO: source). His results show that WebSockets are 31%[38] faster. Therefore, I disagree with his conclusion, as it seems to me that WebSockets outperform Server Sent Events by quite a lot. Another blog post (TODO) showed complete opposite results. This used a proxy with beta stage support for WebSockets, so I don't count these results as reliable.

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

---

[37] Server Sent Events actually beats WebSockets by 0,3 milliseconds. As this is close to 0, I deem them just as good.
[38] WebSockets averaged 374,64 milliseconds. Server Sent Events averaged 490,44 milliseconds. 374,64 + 31% = 490,44.

WebSockets. (TODO: source). 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. 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 this mechanism, also supports streaming over WebSockets. For real time purposes, this is preferable anyways, which renders WebSockets polling little useful. To me though, 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.

10.3 Machine resources

This section will focus on the resource usage of the different transport mechanisms in the context of their framework. Section (TODO) mentions the differences between the various platforms. Implications of this is 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 (TODO) gives an overview of resource usage data collected from the graphs in figure (TODO) and (TODO). 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 use a library for its WebSocket support: Faye (TODO: source), 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 though. It may be something about how Node.js handle this technique. After all, Socket.IO has supported it before, but they do not anymore. (TODO: source).

		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:	70,2%	-	26,2%	-	21,6%	47,8%

	Memory:	888 KB		107 KB		110 KB	209 KB

Another thing that stands out is that Play use more resources than Lightstreamer. As Play is the cleanest implementation, I did not expect this. It shows that you benefit from having a server that only handles real time. Play's server is an application server that also have to serve the MVC application.

It is a trend in the data that the use of machine resources increase as the transport moves farther from WebSockets. (TODO: footer). Except from polling with Lightstreamer it nothing dramatic. Some transports even use a little less of either processor or memory than WebSockets. This difference, though, is so small that it can be counted as equal. This means that Server Sent Events use just as much processor as WebSockets. HTTP-streaming with Lightstreamer also use just as much as WebSockets.

WS-polling use twice as the amount of resources as streaming with WebSockets, 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. (TODO: source). With a 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. In my opinion, this makes a memory leak plausible. It would explain it at least.

Unfortunately, the polling results of Socket.IO cannot be deemed accurate. I have discussed the anomalies before (TODO). The server used less resources, but as the test took 10 seconds longer, the results cannot be compared to Lightstreamer.

10.4Transports effect on network traffic

In section (TODO) I showed figures displaying the captured network traffic and the calculated. (TODO: corssref both). The table below compares the data from the different results. (TODO: table). As described in section (TODO), there were problems with WebSockets, HTTP-streaming and at least long polling with SignalR.

		Polling	Long polling	Streaming	Server Sent Events	WebSockets	WS-polling
SignalR	Calculated: Captured:	-	220M 49M	95M -	94M 45M	65M 45M	-
Socket.IO	Calculated:	105M	104M	-	-	27M	-

	Captured:	31M	30M			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

As long polling and polling don't have an open connection, 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. (TODO: cross). This is a common way of saving network usage, and it is likely that all frameworks do this across all transports.

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.

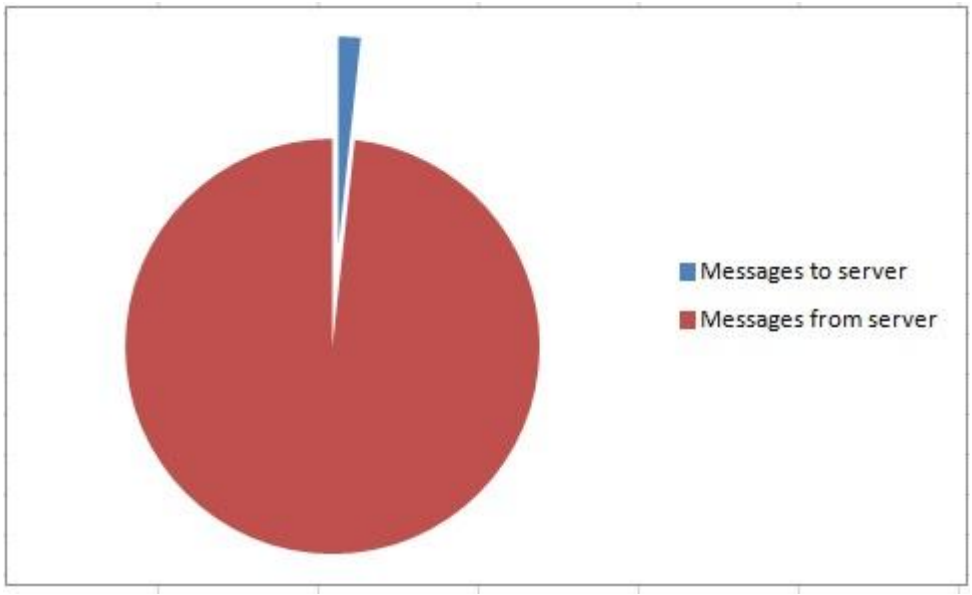
The background chapter (TODO: cross) highlighted overhead regarding header data as a drawback to HTTP. Looking at the result may cause you to believe that this is false for HTTP-Streaming and Server Sent Events. Let us use SignalR as an example. The capture used to calculate throughput contained 14960 bytes for WebSockets and 45879 bytes for long polling[39]. 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. This supports what I wrote initially, 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.

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

[39] The basis for all calculations can be found on GitHub: https://github.com/kjohann/MasterThesis/blob/master/Loadtests/Results/Bytes%20sent%20received%20analysis.md

as it received. In the full test, as you can see from figure (TODO), this is not the case at all. 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.



10.5 WebSockets idle connections resource usage

- TODO: The full picture needs to be on the web
- I use Play to illustrate what is the case with all the frameworks.
- There are some small peaks (2%) every now and then, but that is insignificant. Generally WebSockets uses no CPU what so ever to serve idle connections.

- Idle connections take up memory though. But the increase isn't all that much. Going from 1000 clients to 4500, the memory usage is less than doubled. Overall, WebSockets is very cheap to handle idle connections.
- Of course, with normal request/response HTTP, idle connections cost nothing at all.
- WebSockets is faster than HTTP, uses less overhead when exchanging data, and uses less processor during high loads.
- HTTP has a lot of other things that WebSockets doesn't.
    - Headers
        - Cookies
        - Accept/MIME types (text and bytes are the only options for WS)
    - No issues traversing proxies.
    - No need for a new server.

    - Can do pure push almost as efficient as WebSockets. If you have to upgrade a lot of hardware in order to serve a WS push app, is it worth it?
- Interesting case: WebSockets as basis for new HTTP standard. Would get one connection pr. site, ever..
    - But is it possible? What about headers? Differentiation between get, post, put?
    - Emulating such things with WS creates extra overhead.

Conclusion

11 Frameworks

- The simplest and most enjoyable to work with are SignalR and Socket.IO.
    - Provide an easy to understand model that does not introduce a lot of overhead.
    - Gracefully handles fallbacks, but lets you specify as well.
    - Socket.IO is built according to the mantra of Node.js, providing a lightweight library for real time. Events are easy to relate to and work with.
    - SignalR's use of RPC is clever and intriguing. When you work with it, it actually feels like your making a method call from client to server and vice versa. Lots of magic going on behind.
    - Both offer a more low level abstraction, providing more choice.
    - SignalR is very well documented, with a lot more tutorials than any other framework.
    - Also integrates well with the rest of the ASP.NET stack (IOC, auth).
- Socket.IO shows the best performance regarding latency. Lightstreamer and SignalR follows not far behind.
    - Socket.IO probably doesn't scale as well. But since Node.js is so lightweight you can set up several servers on one machine and still get better performance than IIS or Lightstreamer servers.
    - SignalR sends more bytes than the other two. Also consume more machine resources than the Lightstreamer counterpart.
    - It is very close between them, and they are the best alternatives for their respective language.
- Play Framework is not far from bare metal. But even with some help, it is far more complicated to use than SignalR or Socket.IO.
    - Several things to handle in real time applications
        - Fallbacks
        - Concurrency
        - Clients
        - Reconnect protocol
    - If you only need one transport, it may be beneficial not to use a framework. But since there are simple alternatives, why should you?

12 WebSockets or HTTP?

- WebSockets outperform all HTTP methods when it comes to latency
    - Except SSE
- Resource usage more equal, but there are some clear differences.
    - Especially with polling and long-polling
- Network traffic depends on the nature of the application.
    - Pure push, not too much to gain from WS, but a little.
    - Full bi-directional: a lot to gain.
- Polling and long-polling not very useful compared to HTTP-streaming or SSE.

- WebSockets eligible to take over as HTTP protocol?
    - Lacks constructs for cookies, session data, accept and such.
    - Takes up memory when idle, but not insurmountable amounts. HTTP takes up no resources when idle though.
    - As of now, WebSockets depend on HTTP
        - Would be hard to implement backwards compatibility
        - Moving HTTP aspects over to the WS protocol would pollute the specification. Better to have separation of concerns.
    - Meteor shows how a web application framework can be built with real time in its center. It has a lot of potential. Furthermore, it uses real time for every communication between clients and server – even simple request/response messages. As WebSockets is likely to dominate real time in the future, it seems to me that Meteor's developers believe in the technology for all purposes.
        - Shows that there is nothing wrong with using WebSockets for communication on static web pages.
        - But, I don't see a future where HTML, CSS, images and such are served using WebSockets.

13 Further work

Use this somewhere: There are 1891 messages going from the clients to the server and 108180 messages going the other way. In other words, only 1,7 % of the messages contained in the test are messages to the server.

- Do over, focusing on load testing.
    - Use either headless browser (if WebSocket support) or console clients.
    - Use more clients

Sources

Appendix

Appendix A