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](#_Toc382463302)

[List of figures vii](#_Toc382463303)

[Preface viii](#_Toc382463304)

[Introduction 1](#_Toc382463305)

[1 The World Wide Web 2](#_Toc382463306)

[1.1 HTTP/1.0 2](#_Toc382463307)

[1.2 HTTP/1.1 3](#_Toc382463308)

[1.3 Problem statement 3](#_Toc382463309)

[1.4 Outline 4](#_Toc382463310)

[1.5 Terminology 4](#_Toc382463311)

[Background 5](#_Toc382463312)

[2 Real-time 6](#_Toc382463313)

[2.1 The Real-time Web with HTTP 6](#_Toc382463314)

[2.1.1 Polling 7](#_Toc382463315)

[2.1.2 Long-polling 8](#_Toc382463316)

[2.1.3 HTTP-Streaming 9](#_Toc382463317)

[2.1.4 Comet 9](#_Toc382463318)

[2.1.5 Server-Sent Events 10](#_Toc382463319)

[2.2 WebSockets 10](#_Toc382463320)

[2.2.1 How it works 11](#_Toc382463321)

[2.2.2 The WebSockets API 12](#_Toc382463322)

[2.3 Drawbacks of HTTP techniques 13](#_Toc382463323)

[2.3.1 Really real time? 13](#_Toc382463324)

[2.3.2 When long-polling becomes polling 14](#_Toc382463325)

[2.3.3 Streaming techniques 14](#_Toc382463326)

[2.4 HTTP was never designed for real time 15](#_Toc382463327)

[2.4.1 Overhead 16](#_Toc382463328)

[2.4.2 Half-duplex 16](#_Toc382463329)

[2.5 WebSockets is still young 17](#_Toc382463330)

[2.5.1 Know when to use it 17](#_Toc382463331)

[2.5.2 Know how to use it 18](#_Toc382463332)

[2.6 The use of real time 18](#_Toc382463333)

[2.7 Conclusion 19](#_Toc382463334)

[3 Frameworks for real time web applications 20](#_Toc382463335)

[3.1 SignalR 20](#_Toc382463336)

[3.2 Socket.IO 21](#_Toc382463337)

[3.3 Atmosphere 21](#_Toc382463338)

[3.4 Sails.js 21](#_Toc382463339)

[3.5 Play! Framework 21](#_Toc382463340)

[3.6 SockJS 22](#_Toc382463341)

[3.7 Meteor 22](#_Toc382463342)

[3.8 Lightstreamer 22](#_Toc382463343)

[3.9 Planet Framework 22](#_Toc382463344)

[3.10 XSockets.NET 22](#_Toc382463345)

[4 Servers 22](#_Toc382463346)

[4.1 .NET 23](#_Toc382463347)

[4.2 Java 23](#_Toc382463348)

[4.3 JavaScript 23](#_Toc382463349)

[4.4 Writing your own 24](#_Toc382463350)

[Project part 1: Hands on development 25](#_Toc382463351)

[5 Methodology 26](#_Toc382463352)

[5.1 Selection criteria 26](#_Toc382463353)

[5.1.1 WebSockets support 26](#_Toc382463354)

[5.1.2 Fallback support 26](#_Toc382463355)

[5.1.3 Presentation 26](#_Toc382463356)

[5.1.4 Maturity 26](#_Toc382463357)

[5.2 Description of test application 27](#_Toc382463358)

[5.3 Discussion of use cases 27](#_Toc382463359)

[5.4 Evaluation 28](#_Toc382463360)

[5.4.1 Documentation 28](#_Toc382463361)

[5.4.2 Maintainability 28](#_Toc382463362)

[5.4.3 Simplicity 28](#_Toc382463363)

[5.4.4 Browser support 29](#_Toc382463364)

[5.4.5 Maturity 29](#_Toc382463365)

[5.5 Limitations 29](#_Toc382463366)

[5.5.1 Cloud based solutions 29](#_Toc382463367)

[5.5.2 Rapid development 30](#_Toc382463368)

[5.6 Other choices 30](#_Toc382463369)

[5.6.1 Common UI 30](#_Toc382463370)

[5.6.2 Choice of database engine 30](#_Toc382463371)

[5.7 Selected frameworks 30](#_Toc382463372)

[5.7.1 Socket.IO 30](#_Toc382463373)

[5.7.2 Lightstreamer 31](#_Toc382463374)

[5.7.3 Play Framework 32](#_Toc382463375)

[5.7.4 SignalR 32](#_Toc382463376)

[5.7.5 Meteor 33](#_Toc382463377)

[6 Results 34](#_Toc382463378)

[6.1 Socket.IO 34](#_Toc382463379)

[6.1.1 Documentation 34](#_Toc382463380)

[6.1.2 Simplicity 34](#_Toc382463381)

[6.1.3 Maintainability 36](#_Toc382463382)

[6.1.4 Browser support 37](#_Toc382463383)

[6.1.5 Maturity 37](#_Toc382463384)

[6.2 Lightstreamer 38](#_Toc382463385)

[6.2.1 Documentation 38](#_Toc382463386)

[6.2.2 Simplicity 39](#_Toc382463387)

[6.2.3 Maintainability 42](#_Toc382463388)

[6.2.4 Browser support 43](#_Toc382463389)

[6.2.5 Maturity 43](#_Toc382463390)

[6.3 Play Framework 44](#_Toc382463391)

[6.3.1 Documentation 44](#_Toc382463392)

[6.3.2 Simplicity 44](#_Toc382463393)

[6.3.3 Maintainability 47](#_Toc382463394)

[6.3.4 Browser support 48](#_Toc382463395)

[6.3.5 Maturity 48](#_Toc382463396)

[6.4 SignalR 49](#_Toc382463397)

[6.4.1 Documentation 49](#_Toc382463398)

[6.4.2 Simplicity 50](#_Toc382463399)

[6.4.3 Maintainability 51](#_Toc382463400)

[6.4.4 Browser support 52](#_Toc382463401)

[6.4.5 Maturity 52](#_Toc382463402)

[6.5 Meteor 53](#_Toc382463403)

[6.5.1 Documentation 53](#_Toc382463404)

[6.5.2 Simplicity 54](#_Toc382463405)

[6.5.3 Maintainability 56](#_Toc382463406)

[6.5.4 Browser support 57](#_Toc382463407)

[6.5.5 Maturity 57](#_Toc382463408)

[Project part 2: Load testing 58](#_Toc382463409)

[7 Methodology 59](#_Toc382463410)

[7.1 Test scenario 59](#_Toc382463411)

[7.2 Test data 60](#_Toc382463412)

[7.3 Test setup 60](#_Toc382463413)

[7.4 Choice of setup 61](#_Toc382463414)

[7.5 Displaying data 62](#_Toc382463415)

[7.6 Configurations 62](#_Toc382463416)

[7.7 Monitoring network traffic 63](#_Toc382463417)

[7.8 Monitoring of processor 63](#_Toc382463418)

[7.9 Monitoring of memory usage 63](#_Toc382463419)

[7.10 Limitations 63](#_Toc382463420)

[7.10.1 Meteor 63](#_Toc382463421)

[7.10.2 Using browsers 63](#_Toc382463422)

[7.10.3 Network capture 64](#_Toc382463423)

[7.10.4 Streaming with Play Framework 64](#_Toc382463424)

[8 Results 65](#_Toc382463425)

[8.1 Messages sent from clients 65](#_Toc382463426)

[8.2 Messages received by server 66](#_Toc382463427)

[8.3 Messages sent from server 67](#_Toc382463428)

[8.4 Average latency 67](#_Toc382463429)

[8.4.1 WebSockets 68](#_Toc382463430)

[8.4.2 Server Sent Events 68](#_Toc382463431)

[8.4.3 Http-Streaming 69](#_Toc382463432)

[8.4.4 Long-Polling 69](#_Toc382463433)

[8.4.5 Polling 70](#_Toc382463434)

[8.5 Median processor usage 71](#_Toc382463435)

[8.6 Maximum memory usage 71](#_Toc382463436)

[8.7 Bytes sent/received 72](#_Toc382463437)

[9 Analysis 73](#_Toc382463438)

[9.1 Transports effect on latency 73](#_Toc382463439)

[9.2 Transports effect on machine resources 74](#_Toc382463440)

[9.3 Transports effect on network traffic 75](#_Toc382463441)

[9.4 WebSockets idle connections resource usage 76](#_Toc382463442)

[Conclusion 78](#_Toc382463443)

[10 Frameworks 79](#_Toc382463444)

[11 WebSockets or HTTP? 79](#_Toc382463445)

[12 Further work 80](#_Toc382463446)

[Sources 81](#_Toc382463447)

[Appendix 83](#_Toc382463448)

[Appendix A 84](#_Toc382463449)

# List of figures

# Preface

# Introduction

TODO: Denne skal inneholde en relativt kortfattet oversikt over tema som inngår i oppgaven og til sist en oversikt over hva de forskjellige kapitlene i rapporten inneholder.

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.

## 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?

## Outline

## 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

# Background

# The 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. What started out as a science project is now an important aspect of everyday life.

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, and with HTML5 really starting to make a push, more revolutionary changes are yet to come.

Along with HTML5 comes a new protocol for the Web: WebSockets. It was created to meet one of the newest aspect of web browsing, namely real time applications, where clients can get updates from the server as they occur (TODO: crossref more info section). Real-time web applications has been around for some time, but previously they have relied on the aging HTTP 1.1 protocol.

## HTTP/1.0

Version 1.0 of HTTP was created in the World Wide Web's childhood (TODO: (6): http 1.0). Back then, web pages consisted mostly of text and maybe a few embedded objects[[1]](#footnote-1). But as the Internet grew, and other people than scientists started using it, the need for more vivid content soon became very clear.

At this time, around the mid 90s, CSS too was in its childhood (TODO: (7): Css saga). However, it soon caught people's attention and more and more browsers started to support it (more or less). 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. Getting a HTML-file with a style sheet and three images then required five requests in total, which is obviously inefficient. To circumvent this, some early web applications used several TCP connections at the same time (TODO: (8): Network performance http 1.1). Bear in mind that this was during the old days when download speeds was far from the megabit range.



## HTTP/1.1

Increasing amounts of embedded objects in web pages lead to the creation of HTTP/1.1, which made several vital 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. If the connection to the server was lost half way through the transmission of that particular object, it could later be resumed by using the cached data instead of starting all over. 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 easily could be made backwards compatible with HTTP 1.1. The *upgrade* request-header (TODO: (9): Key differences) makes it possible for a client to request that another protocol should be used if the server supports it.

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 just how much more detailed the 1.1 protocol is[[2]](#footnote-2).

## 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).

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.

# Real-time

As mentioned in (TODO: crossref introduction), 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 automatically update whenever someone posts a comment. An example of an application with more real time content is Facebook, where notifications[[3]](#footnote-3) and your friends’ activities are displayed 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.



## The Real-time Web with HTTP

Recently the concept of real time web has become a buzzword. Having an application pushing information to the client instantly instead of waiting for the client to make a request for it, is how real time application works. However, as we have seen(TODO: crossref background), this is not how HTTP works–the client always has to initiate the communication. To accommodate the growing need for applications of this sort, several techniques have been utilized. Using HTTP in untraditional ways has been the regular way of accomplishing real time (or near real time) until recently, but with the introduction of WebSockets, all of these may be deprecated.

### Polling

As the very first attempt of providing real time updates from a server, polling is a fairly 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, for instance, how to determine the interval to prevent many empty responses and all the same not flooding the server. Therefore, other mechanisms are far more widespread.



There is a way to improve a little upon polling, namely 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 in between the polling interval (TODO: figure 3-2).



### Long-polling

As the name states, Long-Polling is closely 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 to send multiple responds over the same TCP-connection (TODO: figure 3-3). If no new data comes to the server in a given amount of time, the connection normally times out (TODO: (12): A comparison push/pull) and the client reconnects through a new HTTP-request.



### 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 - it remains open throughout the entire course of a client’s session. Service streaming uses a long-lived XMLHttpRequest to send new data, whereas page streaming uses the initial page request. This gives more flexibility regarding the lifetime of the connection.

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) as long as the client is connected, thus using this ability of HTTP 1.1 . Leveraging the fact that a browser executes script-tags[[4]](#footnote-4) whenever it reads them (TODO: (11): Comet and reverse AJAX), the forever frame receives new data from the server wrapped up as such (TODO: figure 3-4). The connection never closes, so each time new data arrives, it is immediately sent to the client and handled appropriately.



### 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[[5]](#footnote-5). With the introduction of WebSockets, it may never be.

### 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 (TODO: (17): Stream updates with..) to push messages to the client without receiving a request first. 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, possibly indefinitely, but at least until it is closed by the client or any intervening proxies. When integrating Server-Sent Events, one can decide how long the connection should stay open and how long it should take before the client reconnects (TODO: (17): Stream updates with..). Server-Sent Events is in other words not too different from long-polling (TODO: figure 3-5).

Unlike long-polling, though, developers using Server-Sent Events have a simple API (TODO: (18): Server Sent Events) that 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 by having the server look up its ID. This makes Server-Sent Events very robust, but is it powerful enough to match its HTML5 brother, WebSockets?



## WebSockets

We have seen that HTTP 1.1, that came only three years after its predecessor, was a significant step ahead. However, since the late 90s, no new HTTP protocol has emerged, even though there are strong indications that the authors believed it would when they made the 1.1 version (see section TODO: http1.1 in background about ). Introducing WebSockets in HTML5 has finally given developers a chance to really make use of the upgrade request-header.

In December 2011, the WebSockets protocol became a proposed IEFT specification under RFC6455 (TODO: (19): WebSockets becomes). The specification document clearly 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)

### How it works

WebSockets, as HTTP, makes use of TCP as underlying protocol. But where HTTP needs several "hacks" (TODO: crossref real time http), WebSockets provides full-duplex communication right out of the box, that makes real time a lot easier.

By having the WebSocket protocol use the same ports as HTTP and HTTPS (80 and 443, respectively)[[6]](#footnote-6), the initial handshake can be done via traditional HTTP (TODO: figure 4-1). The client states that it wants to use WebSockets, and the server sends a response if it supports it[[7]](#footnote-7). Doing it in this way ensures backwards compatibility with older browsers that don't support WebSockets, and allows developers to make their applications fall back to the old HTTP-ways of accomplishing real time.



Sending messages back and forth once the connection is up, is a lot more efficient than what HTTP can provide, and it has a lot less overhead too. Header-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).



### 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 mind, having no support for custom events; just for open, close, receiving a message and error.

Providing an easy way to send messages through the *send* function and an attribute for keeping track of buffered data on the client-side, *bufferedAmount,* the API is rather powerful for developers in spite of being quite simple. The simplicity is, however, 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)

## Drawbacks of HTTP techniques

In section (TODO: real time http), I gave a rudimentary description of different ways to achieve real time, or near 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, mainly because it is supported by even the oldest browsers. However, there are also some issues.

### Really real time?

Long-polling builds upon the idea of polling, but whereas polling is a very naïve approach, long-polling is a lot smarter. One of the major issues with normal polling is how to determine the interval in which the server should be polled.

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. However, this would soon cause any server to crash – unless you have some serious load balancing technology on top, which in turn would lead to a rather expensive solution. Polling the server very often, would also increase the amount of empty responses in cases where data comes to the server in a pulse like manner as shown in (TODO: figure 5-1)on page (TODO: pageref if needed).



How about a longer interval then? Well, with a longer interval, the longer it takes before new data is received, 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, which easily can be retrieved by the client using polling.



### When long-polling becomes polling

As I said, long-polling is a lot smarter than polling. Letting the server keep the request open over a longer period of 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 something there waiting for it that makes the server respond immediately (TODO: (10): Pro Html5) – effectively making long-polling work just as regular polling at a short interval. Comparing (TODO: figure 5-1) to (TODO: figure 5-3), one can clearly see that long-polling does not outperform polling as long as the server-side updates are very frequent.



Norges Bank Investment Management[[8]](#footnote-8) 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, actually fakes real time as it polls the server every 30 seconds and gets the values from the past 30 seconds.

### 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 rather than the client. One could argue that Server Sent-Events isn’t streaming, but it builds upon some of the same ideas as streaming does with its push approach (even though it can be configured to work more like long-polling – see section (TODO: sse section)).

Since the forever frame (section (TODO: forever frame section)) is the far most widespread form of HTTP streaming today, I will focus only on this. While a forever frame allows the server to continuously push updates to the client wrapped up in script-tags, it is far from perfect. Client-side there has to be some extra handling to actually make the received scripts do something useful. Receiving new data in an ever-growing DOM-element, also creates some challenges related to memory management: The frame has to be cleared at regular intervals – otherwise it will take up way too much memory.

Having a persistent HTTP-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). Consequently, many Comet-based streaming solutions, like a forever frame, actually fall back to long-polling when buffering is used.



A forever frame makes the developer write some additional code to handle the incoming scripts. With the EventSource interface of Server-Sent Events, developers have a more powerful toolbox for wrapping the incoming events (see section (TODO: sse section)). Utilizing pure eventhandlers also ensures that there is no need for cleaning up after the incoming data – events are just executed and that’s that. But are there really any major drawbacks to Server-Sent Events? Well, it is still HTTP and as we shall see, the protocol has issues of its own.

## HTTP was never designed for real time

Having introduced the keep-alive flag, chunked encoding and persistent connections in HTTP/1.1(section (TODO: background http 1.1)), one might say that claiming that the protocol wasn’t designed for real time is rather presumptuous. To back up my claim I will look into what I believe to be HTTPs greatest weaknesses compared to WebSockets: its design and, simply, its age.

### Overhead

Previously, 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). In order to get a better picture of why this could be an issue, I will borrow some data from a simple application for comparing polling and WebSockets by Peter Lubbers and Frank Greco (TODO: (23): Benefits of WS). 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. This may not sound like a lot, but when you have clients numbering in hundreds of thousands, the network throughput increases exponentially. A use case with 100 000 users polling every second means that the network in which the server resides, has to deal with 665 megabits per second[[9]](#footnote-9) of throughput. 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[[10]](#footnote-10).

Using polling to represent HTTP against WebSockets is a little unfair in my opinion, seeing how polling is the naïve approach of achieving real time. However, it does prove my point: HTTP-headers have much excess data, but most of the time 99% of this data is completely irrelevant for both server and client. 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.

### Half-duplex

HTTP was finished in the 90s and it is still going strong. It’s actually rather impressive, but it’s also obvious that something that old (and it is really old in computer science terms) will have performance issues towards new trends. WebSockets is a protocol designed solely for the purpose of full-duplex(TODO: (20): WS protocol) communication–HTTP isn’t. In fact, no matter how you look at it, or how you try to hack, HTTP remains half-duplex.

As a result of this, most real time applications with HTTP actually have to use several TCP-connections (TODO: figure 5-5). Even with Server-Sent Events which is the newest invention relying on HTTP, one will need one connection to push the events to the client and at least one more for whenever the client needs to send data back. Recall what I wrote in the background chapter (see section (TODO: crossref http 1.0)) about applications using several TCP-connections with HTTP 1.0 for more concurrent loading of embedded objects; now the same work-around is being repeated to achieve simulated full-duplex communication! And as with last time this was the case, an improvement is needed, namely WebSockets.



## 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 this is being written, 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 natively, and even though IE10, Chrome, Firefox, Opera and Safari does, it will be several years before developers can safely assume that every single client out there supports WebSockets.

Consequently, applications have to fall back to other, supported techniques when WebSocket support is absent, which in turn leads to more code. Luckily, frameworks like SignalR[[11]](#footnote-11) and Socket.io[[12]](#footnote-12) abstract this away for developers, but 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.

### Know when to use it

Writing an application with some real time elements is quite 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.

Using WebSockets for the first example would work excellently, and wouldn’t require too much work either, at least if every client supports WebSockets. But, of course, they do not, leading you as the developer back to workarounds to make it work. You could use a framework, but is it really necessary? Take a step back and analyze what you are going to make. Commenting on a news article is far from chatting, even if it is supposed to show on all clients in real time. In this particular case, the real time aspect of the application is rather small and not that critical for the user experience. Being critical to what your application actually needs to achieve is important in development, and it is easy to be blinded by things that shine brightly like WebSockets does these days.

Chatting is a completely different matter – specifically a chat room, which has several people talking to each other at the same time. This makes real time crucial to the users’ perception of the application, which in turn makes it worth the extra effort of providing fallbacks for the browsers that don’t support WebSockets.

### 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 replace traditional HTTP with WebSockets in applications that don’t really need persistent connections at all. An informative webpage, like Wikipedia, will probably never benefit from using WebSockets. Sure, you get less overhead in request-headers, but on the other hand your application will have to serve mostly idle connections since 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. Though WebSockets is supposed to handle proxies and firewalls gracefully (TODO: (10): Pro Html5), 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://), which, in my opinion, is a good practice since it makes data encrypted.

## The use of real time

The World Wide Web has seen many innovations throughout its lifespan, and 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 even to be noticed, it needs to have some form of establishment throughout the web.

There is no doubt that real time content is very useful in many aspect, and that in others it is even crucial. An auction site with time based auctions completely relies 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 amount of real time content provided by web sites that typically 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 overall 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, though, it soon becomes clear how much of an impact real time might have on our lives in the future. Live video streaming is not a strange phenomenon today, 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.



## Conclusion

We have seen that even though WebSockets is superior to HTTP when it comes to bidirectional communication, it is not always necessary with a full-duplex channel to achieve real time content. 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, it would actually be better to use Server-Sent Events than WebSockets. The need for a fallback for browsers that don’t support this might degrade you to long-polling, which is completely fine as long as the interval in which the server gets updates isn’t too short.

Looking as these aspects leads me to say that HTTP methods may still be a better choice than WebSockets for some real time purposes. However, if we ignore the need for 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. Nevertheless, HTTP, with Server-Sent Events in particular, remains a strong alternative if you only need real time push. Long-polling, HTTP-streaming and definitely polling, I think, will be completely outdated in a couple of years – replaced by WebSockets and some Server-Sent Events applications.

I believe that in the future, when current browsers are considered old and WebSockets has been around for a long while, it will be used in most real time applications. Furthermore, my opinion is that any future versions of HTTP will not incorporate WebSockets – the ywo will remain what they are, namely 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 full-duplex communication applications. It will render HTTP mostly unused for the purpose as soon as the issue of backwards compatibility to clients that don’t support it has vanished. Still, HTTP will remain king of the hill in “traditional” web applications that rely on requesting content in a half-duplex manner.

# 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.

## 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[[13]](#footnote-13).

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)[[14]](#footnote-14) API that allows for simple real time programming.

## 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).

## 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.

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

## 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.

## 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[[15]](#footnote-15) 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.

## 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.

## 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)).

## 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.

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).

## 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.

## XSockets.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.

# Servers

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.

## .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[[16]](#footnote-16), it is a lightweight version of the original. It provides what you normally need while developing, but its performance is not enough for production.

## 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).

## 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).

## 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 ().



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

# 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).

## 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!).

### 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.

### 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.

### 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 useless if you cannot find out how to use it. Lack of proper documentation will make a considerable difference on the negative scale.

### 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.

## 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[[17]](#footnote-17).
* MySql will be utilized as database unless implementing it requires substantial workarounds, that may cause the framework to misbehave.
* The application will be run locally using either the server bundled with the framework, or a server best applicable for it[[18]](#footnote-18).
* All applications should have tests covering the most critical aspects of the program logic.

## 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.

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

### 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.

### 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.

### Simplicity

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

### 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. However, it should not be forced; the user should be able to choose a “lower level” transport[[19]](#footnote-19).

### Maturity

A lot of real time frameworks are a work in progress and has not yet reached version 1.0. Such frameworks often change drastically causing users to change their applications completely to keep up. Hence, these frameworks are not suited for production code–they are not mature enough.

Another way of measuring maturity can be to look at projects that utilizes 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.

## Limitations

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

### 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. Additionally, no cloud vendor would be happy with me running tests against their servers since the tests in practice are the same as a Denial-of-Service attack.

### 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”[[20]](#footnote-20).

## Other choices

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

### 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.

### Choice of database engine

MySql, while it is an aging database engine, is one of the oldest, best maintained and used database engines on the marked. It is reliable and simple to use, and it should be universal enough for all frameworks to use. If, however, some framework does not support it out of the box, I will have to consider not to use it.

Some frameworks may come bundled with a database or not have a complete implementation of MySql-support. If such a framework has been selected for the project, the application will be without MySql. Reasons for why a framework without MySql will be specified.

## 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.

### 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[[21]](#footnote-21)

### 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 experiance plays out against a lot of the new ideas of the more modern frameworks, will be interesting.

Weswit is the name of the company behind Lightstreamer. If I have interpreted their homepage correctly, this 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 thorougly 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[[22]](#footnote-22). In part 2 (TODO: crossref), I used a full version of Lightstreamer.

### 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[[23]](#footnote-23).

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.

### 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.

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.

### 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[[24]](#footnote-24) 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).

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)

# Results

This chapter gives an in-depth description of my experience with each of the frameworks.

## Socket.IO

* Write a summary? Then say the details are in the following sections?

### 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.

### Simplicity

In the spirit of Node[[25]](#footnote-25), 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[[26]](#footnote-26) 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).



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). .



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.

### 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.



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.



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.

### 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).

### 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.

## Lightstreamer

### 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[[27]](#footnote-27). 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 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. Furhtermore, 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 config 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.

### 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).



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.



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\*[[28]](#footnote-28) 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.

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.

### 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.

### 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.

### 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.

## Play Framework

### 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.

### 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)



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.



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

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.



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.



### 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.

### Browser support

Using just WebSockets and Comet provided support for all major browsers[[30]](#footnote-30). 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).



### 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.

## SignalR

### 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[[31]](#footnote-31). The content itself is also separated into logical bulks with lots of examples. A nice feature is that all class names appear as links to the class reference[[32]](#footnote-32).

SignalR is the only framework that covers IOC in its documentation. This is probably because it is the only framework that has its own dependency resolver. It is therefore only natural to tell you 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).

### 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, simple 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.

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).



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[[33]](#footnote-33). 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).



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.

### 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).



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.

### 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.

### 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. This isn't due to the implementation of SignalR though. (TODO: crossref).

## Meteor

* Make a disclaimer somewhere about the version I used contra the one currently available

### 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[[34]](#footnote-34).

### 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.



Meteor encourages the declaration of global level (TODO: footer) 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.

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.



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.

### 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).



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).



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[[35]](#footnote-35), something they wouldn't if it didn't look promising.

### 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).

### 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

# 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.

## 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 og the x-axis in generated graphs. Numbers on the axis represents seconds. 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.

## 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.

More interesting than the message frequency is latency data. Each client needs 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.

## Test setup

There are several ways to implement the test scenario:

* Using console applications as clients.
* Using headless browsers[[36]](#footnote-36).
* Using real browsers.

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 (TODO: example?). 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.

## 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. It would also require different implementations for HTTP and WebSockets, as well as for the different frameworks. 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. The fact that it is not headless yet is also a drawback.

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. Chrome just stopped connecting after six clients had connected, regardless of how many windows they were spread across.

Due to machine resources, I had to limit the tests to only 60 clients running in 30 browser instances. Message frequency was set to two messages per second per client. This load 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"[[37]](#footnote-37) and the "monitor". 19 and 16 tests respectively ensures that these entities perform to specification. In addition, most frameworks needed some extra code to wrap these entities togheter 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).

## 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 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.

## 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.

## 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.

## Monitoring of processor

Procmon?

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. Windows Server 2012’s Resource Monitor was therefore chosen for this purpose.

## 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.

## Limitations

### 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.

### 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. As mentioned before (TODO: setup), the browsers memory consumption, limited the number of clients to 60 for my tests.

### 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. 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 up on GitHub (TODO: link and maybe explain a little? Or make readme..).

### 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 though.

# Results

## Messages sent from clients

* Results are mostly stable at around 120 messages both sent and received pr. second. Chart from WS with sent from clients – display alt 1.
* Some minor drops may be explained by the registering of these events. They may occur on the server (concurrency issues – skipped messages). Another possible explanation is the fact that a browser is single threaded. It handles a lot of incoming events and if a send call is pushed far enough down the call stack, it will result in delayed sending.
* Some deviations with polling and streaming. Chart from Polling with sent from clients.
* Need to discuss plausible reasons why.
  + Exceeded max number of open connections (most probable).
  + The Lightstreamer clients took up a lot more memory than the others (see the connection tests). It may have influenced the ability to send messages with streaming.
  + Max number of open connections most probable. Especially since polling with Lightstreamer did not have the same issues as streaming (two less open connections)



## Messages received by server

* Mostly corresponds with the number of messages sent from the clients. Chart from WS with received by server – display alt. 2.
* The small drops coincides with the drops in messages sent from the clients.



* Lightstreamer shows deviations when using polling. Chart is received by server using polling.
* Most probable cause it that the use of polling causes a lot more stress on the server (see processor usage). This may again lead to the registrations of events happening in bursts.



## Messages sent from server

* Messages sent is registered shortly after received messages. Therefore they should match the values for received multiplied with 60 (the number of clients to broadcast to).
* There are some mismatches though. Chart is a custom with SignalR SSE data for both received at server and sent from server / 60.



* Very small deviations < 0.9 except from the first one. Possible explanation is that my code wasn’t 100% thread safe and that a single send event was lost every now and then. Except for the first one, the messages sent is always less than the messages received, which makes it plausible that one or messages has been lost.
* The deviation with the first message can probably be explained by the high latency number in the first couple of seconds.

## Average latency

* Will show results by transport here – comparing each framework. In the analysis I will compare transports rather than just frameworks.

### WebSockets



* Chart is WS avg. lat.
* I did not run any sort of warm up on SignalR, SockJS or Socket.IO. All of these start off high and then stabilizes on a lower number. As Lightstreamer and Play had warm up, I think that the other three also would have benefitted from this (just not that much).
* Looking at the data from 5 seconds an onwards, it is a clear divide between most of the frameworks.
* Socket.IO has numbers as low as 5 ms, whereas Lightstreamer and SignalR have between 12 and 25 ms. Lightstreamer performs a little better than SignalR.
* Node was designed to be very lightweight. The low load the servers got subjected to in my tests, is probably not enough to show the true scalability. Results gotten by Weswit supports this assumption.

### Server Sent Events



* Chart is SignalR SSE vs SignalR WS
* Only one of the frameworks supported SSE in Firefox. SockJS supported it in Opera, so I have done calculations regarding network traffic for it.
* A very interesting result. SSE actually performs just as well as WS.
* Discuss probable causes in analysis chapter or here? Maybe just put them in one chapter? Instead of results AND analysis?

### Http-Streaming



* Large increase towards the end for Lightstreamer. Probable cause is the implementation of the test coupled with the drop in message frequency towards the end (resulting in overtime).
  + The big question is if this is because of my implementation or something else. Probably my fault though…
* Else, Lightstreamer performs almost ten times better than SockJS. Again, my implementation of the SockJS server may be a cause. It is designed to use Redis to help broadcast, but introducing this new component did not fit my test setup.

### Long-Polling



* SignalR shows a steadily decreasing amount of latency. It is plausible that the server would require a longer warm up period when using long-polling.
  + The fact that Socket.IO increases towards the end while SignalR drops supports this assumption. Both have a small overtime in messages sent from clients (see chart below). All other frameworks and transports, this has resulted in increased latency, not decreased as with SignalR in this case.
  + Still, SignalR is about 3 times slower than Socket.IO in this test, just as with WebSockets.



### Polling



* Socket.IO has a steady latency of a little more than 500 ms. Could have indicated a poll interval of 500 ms, but this is not the case (see bytecalc capture).
  + From the main captures, I can see that multiple messages are put into the same response. This may be so much overhead that it pushes the latency up.
* Lightstreamer had a slight overtime, but the increase in latency is massive towards the end.
  + It shows as much as almost 3.5 seconds, which doesn’t make any sense at all with only a second overtime.
  + Some sort of queue can have occurred within the server, trapping some of the messages sent 2-3 seconds before the end of the test.

## Median processor usage



* How much processor each framework uses is a measure of the server it runs on. Some of the usage may just be background processes that would run no matter what kind of traffic the server experiences (real time or normal).
* Can clearly see the how lightweight Node is.
* Another interesting point is Play vs Lightstreamer. Play, which is very close to a bear metal implementation of WebSockets, uses more processor than Lightstreamer. A display of benefit with using a framework?

## Maximum memory usage



* Yet another display of Node’s power over traditional servers. Using around 30 megabytes of memory throughout the test. NOTE: I saw higher numbers while connecting clients and then sudden drops.
* IIS is probably the most heavy weight server of the bunch, so it is natural that SignalR uses more resources than the others.
* Source of error: 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 was finished. Some of the raw data stands out from the rest because of this.

## Bytes sent/received



* The test data that’s being exchanged is the same across all tests. Hence, this graph actually shows the difference in how much additional data the frameworks put into each message.
* Streaming with SignalR and SSE with SockJS are calculated using different browsers (discussed earlier?).
* Not surprising that Play scores very well as it is close to raw WebSockets. It also does streaming very well.
* More interesting is the way SignalR handles real time: Cursors => quite a lot of extra bytes. Almost twice as much actually.
  + Hard to predict the actual behavior of these cursors.
* A little surprising that Socket.IO actually outperforms Play. Lightstreamer also obviously uses a very good protocol implementation.

# Analysis

## Transports effect on latency



* Cut out the last two-three seconds of the results. Lightstreamer had a sharp increase for all transports but WebSockets in this timeframe (as discussed earlier?). Easier to get a clear comparison with those seconds left out.
* A clear tendency for all frameworks that WebSockets is faster, but the streaming techniques are not far behind.
* Server Sent Events actually performs just the same with SignalR.
  + The test uses a rather low load, so WebSockets may be better with more traffic.
  + SSE is a pure push transport, and this sort of test has more focus on push. 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.
  + All streaming techniques handle pushing of data really well, since the connection used to push is already open.
  + Server Sent Events is, like WebSockets, a HTML 5 API. When keeping that in mind, it really isn’t very surprising that it does push just as well as WebSockets.
* The use of WebSockets to do polling is obviously not meant for a push dominated application. I really don’t see why it is part of the Lightstreamer stack anyways, 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 in order to get data. The result is that the performance drops and streaming over HTTP becomes a better alternative.



* It might seem like long-polling performs really great with Socket.IO, but as WebSockets has average latency around 5 ms, long-polling has more than ten times as much. Looking at (the other graph), one can see that this is also the case with SignalR.
* SockJS is a little different though. Plausible reasons:
  + SockJS uses another library to handle WebSockets (Faye). This may actually make the overall performance of SockJS’s WebSockets.   
    The way I implemented the application may make the benefit of using WebSockets a little less. Since it always has to loop through all clients on the server.
* Polling (over HTTP) seems to be somewhat unstable (other graph). Furthermore, it is far behind the other transports when it comes to latency. Latency can be less with a shorter interval, but then you essentially DDOS the server.
  + On average, it is about ten times slower than WebSockets with Lightstreamer, the same as the difference between long-polling and WebSockets with both Socket.IO and SignalR. This is probably a display of the case when long-polling becomes polling from the background (ref it!). Still, long-polling seems preferable as it appears to be more stable.
  + Socket.IO either has a poor implementation of the polling technique, or it was incompatible with my set up.

## Transports effect on machine resources

* Repeat the charts from earlier? Another reason to not have two different chapers….
* When it comes to processor usage it is reasonable to assume that long-polling and polling uses more than a streaming transport. Both the results of SignalR and Lightstreamer supports this assumption.
* However, looking at the two Node frameworks there are some interesting results.
  + First of all long polling uses about the same as WebSockets with Socket.IO and actually less with SockJS. The lightweight nature of Node may explain this for Socket.IO, but for SockJS it seems that the streaming implementation is more expensive. The reason for the little difference with Socket.IO is probably the relatively low load of the test. It uses around 15% of the total 50% it can theoretically use (single threaded server on a dual core machine).
  + Due to the anomalous behavior of the polling test using Socket.IO, the only data available here that is reliable is for Lightstreamer. Nevertheless, there is a clear indication that handling the extra messages that polling introduces, requires more processor usage. Even when polling over WebSockets, it uses almost twice as much as HTTP-streaming.

## Transports effect on network traffic



* As discussed previously, the recorded network traffic showed some anomalies (errors regarding open connections).
* There are still some interesting points regarding the recorded data versus the calculated. SignalR has a behavior that seems to be hard to predict. This manifests in a very large difference between the number of bytes captured and calculated using long-polling (long-polling and polling (HTTP) should not have been affected).
  + Calculations show more than four times as much data than the captures. But the simple capture used as basis for calculation, shows that the capture with long-polling saw three times as many bytes (ws: 14960, lp: 45879). The calculated data total shows the same (ws: 64,98 M, lp: 219,69 M).
  + There are no cursor messages in the captures for SignalR. This means that they are either ignored or not there when there are a high frequency of messages.
  + For the SSE and Streaming though (and WS-polling), this is not the case. There is only one client in the capture, and it sends only ten messages. That’s ten messages sent and ten received. Using SignalR as an example (looking only at the broadcast and receive which are the dominating messages): 10 x 231 bytes sent and 10 x 597 bytes received = 8280 bytes total. SSE has 10 x 1174 bytes sent and 10 x 845 bytes received = 20190 bytes total, 2,44 times as many bytes. For the real test, there are 1800 messages being sent and 108000 being received. Using the same method, we get 64891800 bytes total for WebSockets and 93373200 bytes total for SSE, only 1,44 times as much. We see that, using streaming, the difference to WebSockets decreases the longer the test runs. The reason for this is the push oriented nature of the test. Sending messages has the greatest difference from WebSockets, as this is done using POST requests, but with these being a minority, the consequence decreases. This effect does not apply to long-polling nor polling, since these techniques uses a GET og POST request in order to receive data from the server.
* Why there is such a difference between the recorded data and the calculated, I really don’t know. It may be that each framework handles a lot of messages more effectively than just a few. The use of two clients in one browser, may also have had some repercussions that are beyond my comprehension.
* TODO: how can I write that I really don’t understand something? Because I really don’t.
* Despite the differences between the captured data and the calculated data, I believe that the latter gives the more accurate description. Even though it is theoretical, it is based on the same grounds. In my opinion the most accurate representation of the differences between transports is the calculated data.

## 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

# 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?

# 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.

# Further work

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

# Sources

# Appendix

# Appendix A

1. Embedded objects consisted mostly of images, but also some early forms of style sheets. [↑](#footnote-ref-1)
2. 56 vs. 162 pages when copied as they are from [http://www.ietf.org](http://www.ietf.org/) into Microsoft Word. [↑](#footnote-ref-2)
3. You receive a notification whenever someone likes or comment on an item that is somehow related to your profile (tags, mentioning your name, etc.). See [www.facebook.com](http://www.facebook.com). [↑](#footnote-ref-3)
4. The forever frame receives JavaScript code wrapped up in script-tags. [↑](#footnote-ref-4)
5. Internet Engineering Task Force - Request for Comment series: see http://www.rfc-editor.org/ [↑](#footnote-ref-5)
6. The WebSocket counterparts are ws and wss. [↑](#footnote-ref-6)
7. Status code 101 [↑](#footnote-ref-7)
8. <http://www.nbim.no> [↑](#footnote-ref-8)
9. 87 100 000 bytes \* 8 = 696 800 000 bits / 10242 = 665 Mbits [↑](#footnote-ref-9)
10. 200 000 bytes \* 8 = 1 600 000 bits / 10242 = 1.526 Mbits [↑](#footnote-ref-10)
11. <http://signalr.net/> [↑](#footnote-ref-11)
12. <http://socket.io/> [↑](#footnote-ref-12)
13. In order to write C# code on Android and iPhone, one has to use the cross platform framework Xamarin (https://xamarin.com/). [↑](#footnote-ref-13)
14. 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). [↑](#footnote-ref-14)
15. 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). [↑](#footnote-ref-15)
16. Visual Studio Express is one of these products. [↑](#footnote-ref-16)
17. [www.knockoutjs.com](http://www.knockoutjs.com) [↑](#footnote-ref-17)
18. For instance a framework for .NET is natural to run using IIS Express. [↑](#footnote-ref-18)
19. WebSockets is the highest level possible, whereas polling is the lowest. [↑](#footnote-ref-19)
20. https://github.com/Atmosphere/atmosphere [↑](#footnote-ref-20)
21. 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) [↑](#footnote-ref-21)
22. See full list of features here: <http://www.lightstreamer.com/products> [↑](#footnote-ref-22)
23. See Play’s homepage at the bottom: <http://www.playframework.com/> [↑](#footnote-ref-23)
24. 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. [↑](#footnote-ref-24)
25. According to Node.js’s homepage (TODO: source), its event-driven model makes it lightweight and efficient. [↑](#footnote-ref-25)
26. [www.ign.com/?setccpref=US](http://www.ign.com/?setccpref=US) (American IGN). [↑](#footnote-ref-26)
27. The subscription modes are RAW, MERGE, COMMAND and DISTINCT. [↑](#footnote-ref-27)
28. MVVM (Model View ViewModel), MVC (Model View Controller) and MVP (Model View Pattern) are referred to as MV\*. [↑](#footnote-ref-28)
29. An example of an absolute path to a function is: window.someGlobalObject.someGlobalFunction(). [↑](#footnote-ref-29)
30. I tested with Internet Explorer 8 and newer and the newest versions of Chrome, Firefox and Opera. [↑](#footnote-ref-30)
31. There is one documentation for the 1.x versions and another for 2.x. Play has more separation, with documentation for both 2.0, 2.1.1 and 2.2.0 for example. [↑](#footnote-ref-31)
32. Class references in C# are similar to Javadocs for Java. [↑](#footnote-ref-32)
33. 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. [↑](#footnote-ref-33)
34. You can see one of the videos here: <https://www.meteor.com/screencast> [↑](#footnote-ref-34)
35. A quick Google search reveals a lot of articles and blogs about Meteor. This indicates that Meteor has the attention of many people in the web development community. [↑](#footnote-ref-35)
36. A headless browser is a browser without the graphical user interface (TODO: what is a headless browser - blog). [↑](#footnote-ref-36)
37. I implemented the setup with SignalR first. I therefore chose to use its terminology. [↑](#footnote-ref-37)