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](#_Toc381799495)

[List of figures vii](#_Toc381799496)

[Preface viii](#_Toc381799497)

[Introduction 1](#_Toc381799498)

[1 The World Wide Web 2](#_Toc381799499)

[1.1 HTTP/1.0 2](#_Toc381799500)

[1.2 HTTP/1.1 3](#_Toc381799501)

[1.3 Problem statement 4](#_Toc381799502)

[1.4 Outline 4](#_Toc381799503)

[1.5 Terminology 4](#_Toc381799504)

[Background 5](#_Toc381799505)

[2 Real-time 6](#_Toc381799506)

[2.1 The Real-time Web with HTTP 6](#_Toc381799507)

[2.1.1 Polling 7](#_Toc381799508)

[2.1.2 Long-polling 8](#_Toc381799509)

[2.1.3 HTTP-Streaming 9](#_Toc381799510)

[2.1.4 Comet 9](#_Toc381799511)

[2.1.5 Server-Sent Events 10](#_Toc381799512)

[2.2 WebSockets 10](#_Toc381799513)

[2.2.1 How it works 11](#_Toc381799514)

[2.2.2 The WebSockets API 12](#_Toc381799515)

[2.3 Drawbacks of HTTP techniques 13](#_Toc381799516)

[2.3.1 Really real time? 13](#_Toc381799517)

[2.3.2 When long-polling becomes polling 14](#_Toc381799518)

[2.3.3 Streaming techniques 14](#_Toc381799519)

[2.4 HTTP was never designed for real time 15](#_Toc381799520)

[2.4.1 Overhead 16](#_Toc381799521)

[2.4.2 Half-duplex 16](#_Toc381799522)

[2.5 WebSockets is still young 17](#_Toc381799523)

[2.5.1 Know when to use it 17](#_Toc381799524)

[2.5.2 Know how to use it 18](#_Toc381799525)

[2.6 The use of real time 18](#_Toc381799526)

[2.7 Conclusion 19](#_Toc381799527)

[3 Frameworks for real time web applications 20](#_Toc381799528)

[3.1 SignalR 20](#_Toc381799529)

[3.2 Socket.IO 20](#_Toc381799530)

[3.3 Atmosphere 20](#_Toc381799531)

[3.4 Sails 20](#_Toc381799532)

[3.5 Play Framework 20](#_Toc381799533)

[3.6 SockJS 21](#_Toc381799534)

[3.7 Meteor 21](#_Toc381799535)

[3.8 Lightstreamer 21](#_Toc381799536)

[3.9 Planet Framework 21](#_Toc381799537)

[3.10 Netty 21](#_Toc381799538)

[4 Servers 21](#_Toc381799539)

[4.1 .NET 21](#_Toc381799540)

[4.2 Java 21](#_Toc381799541)

[4.3 JavaScript 21](#_Toc381799542)

[Project part 1: Hands on development 22](#_Toc381799543)

[5 Methodology 23](#_Toc381799544)

[5.1 Selection criteria 23](#_Toc381799545)

[5.1.1 WebSockets support 23](#_Toc381799546)

[5.1.2 Fallback support 23](#_Toc381799547)

[5.1.3 Presentation 23](#_Toc381799548)

[5.1.4 Maturity 23](#_Toc381799549)

[5.2 Description of test application 24](#_Toc381799550)

[5.3 Discussion of use cases 24](#_Toc381799551)

[5.4 Evaluation 25](#_Toc381799552)

[5.4.1 Documentation 25](#_Toc381799553)

[5.4.2 Maintainability 25](#_Toc381799554)

[5.4.3 Simplicity 25](#_Toc381799555)

[5.4.4 Browser support 26](#_Toc381799556)

[5.4.5 Maturity 26](#_Toc381799557)

[5.5 Limitations 26](#_Toc381799558)

[5.5.1 Cloud based solutions 26](#_Toc381799559)

[5.5.2 Rapid development 27](#_Toc381799560)

[5.6 Other choices 27](#_Toc381799561)

[5.6.1 Common UI 27](#_Toc381799562)

[5.6.2 Choice of database engine 27](#_Toc381799563)

[5.7 Selected frameworks 27](#_Toc381799564)

[5.7.1 Socket.IO 27](#_Toc381799565)

[5.7.2 Lightstreamer 28](#_Toc381799566)

[5.7.3 Play Framework 29](#_Toc381799567)

[5.7.4 SignalR 29](#_Toc381799568)

[5.7.5 Meteor 30](#_Toc381799569)

[6 Results 31](#_Toc381799570)

[6.1 Socket.IO 31](#_Toc381799571)

[6.1.1 Documentation 31](#_Toc381799572)

[6.1.2 Simplicity 31](#_Toc381799573)

[6.1.3 Maintainability 32](#_Toc381799574)

[6.1.4 Browser support 33](#_Toc381799575)

[6.1.5 Maturity 33](#_Toc381799576)

[6.2 Lightstreamer 34](#_Toc381799577)

[6.2.1 Documentation 34](#_Toc381799578)

[6.2.2 Simplicity 35](#_Toc381799579)

[6.2.3 Maintainability 37](#_Toc381799580)

[6.2.4 Browser support 38](#_Toc381799581)

[6.2.5 Maturity 38](#_Toc381799582)

[6.3 Play Framework 38](#_Toc381799583)

[6.3.1 Documentation 38](#_Toc381799584)

[6.3.2 Simplicity 39](#_Toc381799585)

[6.3.3 Maintainability 41](#_Toc381799586)

[6.3.4 Browser support 42](#_Toc381799587)

[6.3.5 Maturity 42](#_Toc381799588)

[6.4 SignalR 43](#_Toc381799589)

[6.4.1 Documentation 43](#_Toc381799590)

[6.4.2 Simplicity 43](#_Toc381799591)

[6.4.3 Maintainability 45](#_Toc381799592)

[6.4.4 Browser support 46](#_Toc381799593)

[6.4.5 Maturity 46](#_Toc381799594)

[6.5 Meteor 46](#_Toc381799595)

[6.5.1 Documentation 46](#_Toc381799596)

[6.5.2 Simplicity 47](#_Toc381799597)

[6.5.3 Maintainability 48](#_Toc381799598)

[6.5.4 Browser support 49](#_Toc381799599)

[6.5.5 Maturity 49](#_Toc381799600)

[Project part 2: Load testing 51](#_Toc381799601)

[7 Methodology 52](#_Toc381799602)

[7.1 Test scenario 52](#_Toc381799603)

[7.2 Test data 52](#_Toc381799604)

[7.3 Test setup 52](#_Toc381799605)

[7.4 Choice of setup 52](#_Toc381799606)

[7.5 Monitoring network traffic 53](#_Toc381799607)

[7.6 Monitoring of processor 53](#_Toc381799608)

[7.7 Monitoring of memory usage 53](#_Toc381799609)

[7.8 Limitations 53](#_Toc381799610)

[7.8.1 Meteor 53](#_Toc381799611)

[7.8.2 Using browsers 53](#_Toc381799612)

[7.8.3 Network capture 53](#_Toc381799613)

[8 Results 54](#_Toc381799614)

[8.1 Messages sent from clients 54](#_Toc381799615)

[8.2 Messages received by server 55](#_Toc381799616)

[8.3 Messages sent from server 56](#_Toc381799617)

[8.4 Average latency 56](#_Toc381799618)

[8.4.1 WebSockets 57](#_Toc381799619)

[8.4.2 Server Sent Events 57](#_Toc381799620)

[8.4.3 Http-Streaming 58](#_Toc381799621)

[8.4.4 Long-Polling 58](#_Toc381799622)

[8.4.5 Polling 59](#_Toc381799623)

[8.5 Median processor usage 60](#_Toc381799624)

[8.6 Maximum memory usage 60](#_Toc381799625)

[8.7 Bytes sent/received 61](#_Toc381799626)

[9 Analysis 62](#_Toc381799627)

[9.1 Transports effect on latency 62](#_Toc381799628)

[9.2 Transports effect on machine resources 63](#_Toc381799629)

[9.3 Transports effect on network traffic 64](#_Toc381799630)

[9.4 WebSockets idle connections resource usage 65](#_Toc381799631)

[Conclusion 67](#_Toc381799632)

[10 Frameworks 68](#_Toc381799633)

[11 WebSockets or HTTP? 68](#_Toc381799634)

[12 Further work 68](#_Toc381799635)

[Sources 69](#_Toc381799636)

[Appendix 71](#_Toc381799637)

[Appendix A 72](#_Toc381799638)

# List of figures

# Preface

# Introduction

# The World Wide Web

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 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 over the same TCP connection (TODO: (8): Network performance http 1.1)., and it was a dramatic change at the time, as it gave allowed clients to get several objects in one request.

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). Regardless of the advance HTTP 1.1 was, the next step in internet evolution may prove to be even bigger.

## Problem statement

In this thesis, I will look at real time applications in general, and WebSockets in particular, and investigate how much of an impact this new protocol can make on the real time world. I will also look at the bigger picture, and investigate the necessity of real time.

Furthermore, I will compare five different frameworks for real time web applications based on both usability, from a programmers perspective, and performance through load testing. A detailed description of how this will be executed can be found in chapter (TODO: crossref Methodology).

The results will consist of my evaluations of the frameworks, charts and discussions about the performance tests and a general discussion that compares WebSockets with HTTP. Furthermore, I will assess whether or not a framework is necessary at all for building scalable real time applications across multiple platforms (browsers).

## 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

# 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 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. I will use this fork for my test. The only real difference is the command line tool that has te 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). .



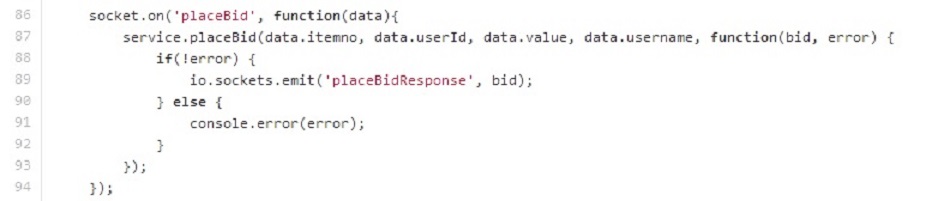
Because of this simple abstraction, sending a message never require more than a single line of code. Socket.IO also provides constructs for sending messages to a specific client. To do this you need the id of the particular client's socket. Associating this with for instance a username, has to be handled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

* The installation process mostly consist of downloading and extracting a .zip archive. This process is very well documented.
* Massive documentation – a clear indication that this is a framework rather than a library.
* Very well organized into several, smaller document, one for each API.
  + Rather strange that the documentation is so comprehensive, but it offers little insight into the most central aspects of the framework: the different subscription modes.
  + I still don’t feel that I have a proper understanding of these. Best explanation I found was in the forums.
* Very few tutorials or detailed examples (actually only one).
* Points you to the demos that ship along with the framework.



* + These are documented through comments (few).
  + Not clear what parts of the code that regards UI and what regards updates.
* There are also quite a lot of documentation regarding the various settings of the server.
  + In the config file!
  + Should be on a more general level in a separate document.

### Simplicity

* A huge framework that does a lot more than just provide real time.
* Is meant to be used as a stand-alone part of an application stack.
  + A separate server rather than just a layer above a normal application stack.



* Offers session management, authorization for the Lightstreamer server. It needs to do this, as it is meant to be a stand-alone service.
* Built kind of like a SOA with a publish/subscribe model.
  + Doesn’t follow all SOA principles (TODO: source).
    - Is not discoverable.
    - There is a somewhat strong coupling. There can be several, independent data adapters, but only one metadata adapter.
  + Still, the LS server functions optimally if it can function as a publisher (in SOA terms). Some other part of the application stack would then be producers, while the clients are consumers.
* Clients subscribe to different items, rather than registering listeners for certain events as all the other frameworks does.
  + To me, this feels a little old fashioned, and it creates a very tight coupling between the DOM and the available items.
  + Each item field has to correspond to a field of an item.



* There are ways to work around this by using only one DOM element for each item, and sending JSON data.
  + Have to handle event routing manually. Also have to serialize manually.
    - Data serialization is manual no matter, but converting to JSON adds another layer of complexity.
    - It also makes you unable to benefit from the different subscription modes.



* + Not the way Lightstreamer is intended to be used.
* The different subscription modes are smart, but complex. Again, there is tight coupling to the DOM.
* Using Lightstreamer with client side MV\* frameworks like Angular or Knockout is impossible.
* Even in its simplest form, Lightstreamer is the most complicated framework.
  + Metadata adapter handles incoming messages (also sessions, auth and such)
  + Has to direct all incoming traffic yourself – no mechanism.
  + Need to send to the data adapter for pushing to the client.
  + Concurrency has to be handled manually through the use of event listeners that you have to make.



* Lightstreamer is broadcast by default. Being so old, it is obvious that the main use is, and always has been for push applications.
* To send to individual clients, even back to the caller, one has to have own subscriptions for these.

### Maintainability

* Despite the complexity, there is little problem writing maintainable code.
* Testing methods that utilize an execution listener can get a little messy, but it is possible. Need IOC for the listener.
* My application never used any of the data that gets injected into the adapters at init, but if an application does this, it is easy to inject some dummy data in a test.
* Built with maintainability in mind. But developers need to do an extra effort to keep code that way compared to the other frameworks.
* Client side is an whole other story.
  + The client side library is very advanced, and I have only been able to get a minified version of the JavaScript file.
  + This gives less insight into an entity that you would want to stub or mock in the client side tests.
  + Most likely to use only functional tests to cover the client. A solution that works. Ideally one would want to use functional tests to accompany other tests as well.
* Keeping the Lightstreamer logic separated from all other code in an application stack helps maintainability.
  + Less likely to mix up code at least.
  + Still, with the other frameworks, there would be one folder with one or more files regarding the real time specific code.

### Browser support

* Supports all major browsers.
* Really cannot understand the need for polling via WebSockets though.

### Maturity

* Released in 2000!
* Can tell that it is aging.
* Lightstreamer is a completely different approach than any of the competition.
  + Keeping it on a separate server may be a good thing. Then it is not up to the application server to throttle bandwidth. On the other hand, it adds an extra security layer if the application needs to be secure throughout the stack (which most do).
* Weswit has a lot of experience in the real time business, being one of the pioneers for the technology.
  + A little surprising that they haven’t made a “Lightstreamer light”.
    - Should be possible to use the same core with a higher abstraction level and less complexity.
    - This could allow for a SignalR like framework, still backed by the Lightstreamer server. I believe this would make the framework seem more attractive for Java developers that want some simple real time features in their application.
  + The company has a large customer base, and has received many good reviews.
    - “If it ain’t broken, don’t fix it”
    - For some use cases, I feel that it is broken.

## Play Framework

### Documentation

* Simple documentation for the simple process of downloading and installing (unzip).
* Scala and Java versions have completely separate documentation.
  + Except for the getting started part and other aspects that are language independent.
  + Allows Java developers to focus on Java and vice versa. Looking at examples that are not applicable for your current work, makes the overall process less appealing.
* Keeps version history of the API documentation.
* Some issues with broken links. Even in some of the main introductory tutorials, there are links to vital parts that are broken.
* Play builds on some third party libraries.
  + Not described in too much detail in the Play docs, but rather points the reader in the right direction. Allows for further reading by need.
* Example based rocks!

### Simplicity

* Write a little about the MVC aspects, but focus on the real time.
  + Quite standard with a route configuration file that is easy to understand. It even gives you feedback and compilation errors.
    - Have to manually assign all routes though.
  + Uses Scala as template language, but not on a very high level. Feels like Razor (.NET).
* Builds upon Akka’s actor model.
  + Unfamiliar, but an increasingly popular programming model for event driven architechtures.
  + Don’t have to use it, as with everything else with Play.
* Offers utility classes for WebSockets and streaming (Comet)
  + Almost no abstraction.
  + The WebSocket class gives you an “in”-channel and another for “out”.



* + The in channel gives you access to the onclose and onmessage events of the WebSocket API, while the WebSocket class itself has the onconnect event.
  + Messages are sent using the out channel.
  + The Comet class provides a similar API except for the in channel.
    - Incoming messages has to be handled by ordinary request to the server. POST messages are preferred for this type of messages.
    - Provides a callback to handle disconnects, which is very handy.



* + Using Comet, one has to have an iframe on the client that can receive outgoing messages. These are wrapped in script-tags by the utility class. Have to provide the Comet class with the name of the function to call on the client. This function either have to be located in the global scope, or you have to provide a specific path (TODO: footnote with window.something.somefunc).
* Utility classes can be based either on strings or on Jackson JsonNode. Play uses Jackson as internal JSON handler already, so this is what feels natural.
* Can easily get request body from a POST request using a simple utility method.
* Using Jackson feels somewhat like Lightstreamer’s use of maps. It is basically the same. Arrays add an extra layer of complexity though, as you have to make an ArrayNode structure manually



* Easy to lose track if the data model is complex enough. Nesting a lot of arrays require a lot of looping through data. Overall, this have negative effects on performance.
* One of the reasons for choosing Play was to get some close to bear metal experience regarding real time applications (TODO: crossref method).
  + Have to use a hashmap or some other list structure to maintain clients. Hashmaps are probably the most effective.
  + Need to have a reliable source of an unique id for each client. Play gives you a connection id that can be used for this purpose.
  + In order for this to work, the WebSocket and Comet classes has to be wrapped in another class that inherit from the same superclass.



* Sending a message is the only different code, so it does not add a lot of overhead. Implementing long-polling though, would probably have been a whole other story.

### Maintainability

* Mostly simple to write maintainable code, but there are some issues:
  + The utility classes for WebSocket and Comet are both event based. How to get to those events?
    - Same problems as with Socket.IO really. Simplest solution is to do the same as with Socket.IO and have each event simply calling a method in another class than can be tested.
  + Using Actors are also hard to test and maintain.
    - Possible to build huge hierarchies of actors that work together.
    - Testing an actor is not straight forward, but again, the above solution is a simple workaround.
* Still need some way to test that everything works together.
  + Play offers some test utilities to help with this.
    - Start a fake server and use Selenium to do functional tests.
      * Had issues with this regarding actors. Also hard to debug.
      * Does one really need this?
      * Normal to have a continuous integration environment, with a server already running. Can test towards this.
      * Also gives a more correct image of the whole application. Play also offers an in memory database you can use with each test. This ensures that you have a fresh data set each test, but the overall overhead isn’t necessarily reduced. Need to do the same for a functional test towards a server running somewhere in the environment.
    - Has other neat utilities to help test things that doesn’t require a running server. Controllers, rendering of views and such.
    - The H2 database is very neat to use if you write your models as EBeans. This eliminates the dao layer, but even with daos, using the H2 database is preferred. Still need the higher level test.
* Using Actors are not required. Even if it was, it is perfectly possible to write highly maintainable code using them. Otherwise, the pattern would not have gained such popularity.

### Browser support

* Using just WebSockets and Comet provided support for all major browsers
  + IE >= 8 and newest versions of Chrome, Firefox and Opera tested.
* Handling fallback manually can be achieved in two different ways. I chose to use the user agent string to determine what browser it was.



* Not the best solution in retrospect.
  + Can 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

* Introduced in 2007, but Play as it is today in 2012 (2.0).
* 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 no matter what language you utilize. However, this also means that the core is quite new and unproven. Furthermore, Scala is a relatively new language. Al though it’s popularity is almost increasing by the minute, it still isn’t used that much. The TIOBE Programming Community Index[[27]](#footnote-27), 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 mostly my opinion as well, but there was some issues like outdated versions of third party software and broken links on the homepage (TODO: crossref). In addition, there are some known bugs regarding functional tests using the fake server utility (TODO: crossref testing or source). All these issues, no matter how small, seems like childhood kinks that a fully matured framework shouldn’t have.
* Offers a lot of freedom. CoffeeScript LESS etc..
* Does not have the same stability as for instance ASP.NET and definitely not the same maturity as Spring MVC. However, the latter is on retreat, while Play is on the rise.

## SignalR

### Documentation

* Has a very comprehensive getting started tutorial. Almost overkill for the simple installation process (nuget).
* During early development, the only documentation was on GitHub
  + Quite extensive nonetheless though.
  + Had no tutorials, but the documentation was based on examples.
* The new documentation has a lot of tutorials and examples and is way more comprehensive.
  + Separated nicely. Both on version level and content.
  + Has kept the example based approach.
  + Links to class references via the code examples. Very nice!
* Also provides examples for IOC, something I really missed with every other framework.
* Hubs are kind of tightly coupled with the mechanisms that handle sending messages. There should have been some documentation regarding making hubs testable.
* No direct links to persistent connections in the documentation. Only on git.

### Simplicity

* Built to be integrated into an existing ASP.NET application stack.
  + Meant to be compatible with authentication, IOC and session management.
  + A developer only needs to find out how to match those things with SignalR. This task is made rather simple, and there are examples for almost everything.
* SignalR comes in two different forms: Hubs and persistent connections.
  + I have focused on hubs, as this is the most common.
  + Persistent connections are a lower level of abstraction over the transports.
  + More like the WebSockets API but with additional methods for handling sending, broadcasting, reconnection, groups etc..
  + Allows you to make for instance a standard MVC action method that broadcasts data to connected users.
* The Hub API strives to be very simple, providing a RPC (Remote Procedure Call) model which makes the code very simple and understandable.
* All public methods are exposed to the client and can be “called” directly.



* The only source of confusion is the different naming conventions of C# and JavaScript. C# has all method names starting with an upper case letter, while JavaScript uses a small case letter. If you are not used to this, it doesn’t take too long to grow accustomed to.
* Clients are managed through an abstraction you can really relate to and that makes sense.
  + Can just return from a method -> return back to the caller, a call from the client returns a promise, just like a normal $.get/$.post and so on.
  + Can also specify functions for the client that are callable from the server in the same RPC style.



* + The Clients object also has construct to handle caller (to send response via another function than via the promise), groups (namespaces)
* Serialization is done automatically behind the scenes using JSON.
  + Even date objects are handled. I ran into a problem with this though, as it deserialized dates with wrong time. This turned out to be UTC time instead of GMT + 1 which is the time zone my machine has. Problems with date therefore persists, even though it seems like SignalR has solved it at first glance.

### Maintainability

* SignalR is just a real time layer for a normal web application.
* Compared to any other web application with ASP.NET, you only have one extra entity to test: the Hubs (or persistent connections if you’re using that).
* A Hub is made with testability in mind, but you cannot test a hub class directly.
  + This is not uncommon.
  + Have do derive from the hub you want to test and make a testable hub (TODO: ref: http://software.intel.com/en-us/blogs/2013/09/25/unit-testing-on-signalr-hubs-with-20-rc1).
  + Setting up the tests like this allows you to verify all logic within a hub.



* + The site also claims that you can test that you get the value of whatever was sent to the client as well, but I was not successful in doing that.
    - It is more normal to test such matters through functional tests anyways.

### Browser support

* Supports all major browsers
* You can, as with all the other frameworks choose what transport to use, but 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

* SignalR has been in the works for a couple of years, but didn’t reach 1.0 until early 2013 (TODO: ref releases git).
* Updates to both 1.x.x and 2.x.x versions are coming frequently.
* Maintained now more than ever – backed by a substantial team.
* Has become an integrated part of the ASP.NET framework
  + Even used by Visual Studio!
* Focuses a lot of attention to promoting SignalR at conferences.
  + Microsoft really believes in the technology.
* After reaching 1.0, the code platform will most likely not undergo any serious changes. It is very stable.
* A little sad that you must use IIS8 on Windows 8+ or Windows Server 2012+ in order to utilize WebSockets. This is not SignalR’s fault though.

## Meteor

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

### Documentation

* Download and installation process described in documentation. For Linux/Mac only though, which is natural…
* Even though they have a roadmap, I find it a little weird that there’s no mention of Windows at all in the getting started section. Could have referenced the roadmap there.
* Meteor is a work in progress – also the docs
  + Constantly changing and makes it hard to follow sometimes. What version does this apply to?
  + I knew these things when I chose Meteor, and it is expected of a framework that is not completed.
  + Could have used something like Node’s documentation though.



* There are a few examples and screencasts, but no tutorials.
* Actually an impressive amount of documentation for something that is a long way from finished.
* Example oriented, which is very good.

### Simplicity

* The motivation behind Meteor is simplicity.
* Right now, it is not that simple in all aspects.
  + Especially since the package aspect wasn’t as complete as it is now when I worked with it.
* A little confusing with all the magic going on in the background.
  + Automatically connected to real time.
  + Bundling and wrapping of code.



* + Declaring code at a global level is encouraged when working with Meteor, but it isn’t always clear what the effect of this will be. In the above example, the function “addItem” ends up in the global scope. Polluting the global scope is strongly discouraged when working with JavaScript (TODO: source needed?).
  + Meteor has introduced namespacing (TODO: soruce docs). Each package gets its own global scope.
* Files that are dependent on each other also becomes a little tricky based on the automation Meteor introduces. It loads files based on where they are in the file system.
  + Impacts how you structure your application.
  + Forces you to make extra folders to get something deep enough to be loaded first.
* Really weird that all changes to the global data will be sent to all users.
* No simple messaging mechanism. Has to store a message within a database.
* Global access to all published records in the database (auto is on by default).
  + A cool feature that you can do inserts and updates directly on the client.
  + Lots of security issues.
  + Have introduced authentication (TODO: source screencast). Looks weird with the snap-back of the UI. May be able to prevent that though by declaring client specific code for what to do when database entries are being updated.
* Benefit of sharing a lot of code is that there is no serialization going on in the open. All is behind the scenes.
  + Though with Meteor, I feel that way too much is behind the scenes. Half the time I don’t know what’s going on.

### Maintainability

* There is no official testing framework as of know. A little disturbing that official testing framework is stated as “In 1.0 if time permits”.
* A framework built on tight couplings makes a lot of alarms go off in my head.
* There is not clear concept of a unit to me with Meteor, maybe due to the coupling.
* Exists ways to test everything though.
* My approach was rather simple. Load it up on Node with Mocha and use a simple library to inject mocks into the files under test.



* Laika is an interesting framework that I unfortunately didn’t get to work on Windows. But it proves that testing Meteor apps is perfectly feasible.
  + Will be integration tests though.
  + Even with a separate database that resets between each test.
* The way Meteor is build makes integration tests a lot more prominent than unit tests.
* Cannot pass judgement before the framework is finished, but from what I can see now, one will have to be very strict with the way applications are structured. Otherwise, it is a short way from heaven to a confused hell.

### Browser support

* Supports all major browsers.
* A small quirk regarding SockJS is that it has to point to 127.0.0.1 instead of localhost in order to use WebSockets..

### Maturity

* An open source framework with steady development since 2011.
* Has recently started to receive appraisal from many developers, boosting the community.
* Received a rather large financial backing in 2012 (TODO: receives funding), assuring that some developers can work on it full time (TODO: meteor/about/people).
* Is a very new way of thinking, something that has never been done before.
  + Can change the way we think of web applications.
  + Will probably not be a universal framework suitable for every single task.
  + Promises a lot, but may have taken on too much?
* 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 than migrating to a newer version potentially can force you to rewrite large portions of your application, which is completely unacceptable for any (or at least almost any) software company.
* Can work a little on the presentation of the homepage…

# Project part 2: Load testing

# Methodology

* Each framework from part 1 will undergo load testing to determine:
  + Which framework has the best performance
  + Is WebSockets really that much better than HTTP?

## Test scenario

* Describe how the tests will be performed in a general way
  + A number of clients sending messages to a server
    - Describe message flow: initTest -> initTest -> broadcast…
  + Describe what data will be recorded and how.

## Test data

* Describe what data to collect and how.
  + Mention time issues and why there is a separation between server start and client start.
  + Discuss probable causes of error regarding data collection
    - Having each client calculating latency based on when it receives a response to its own message.
    - Registering sent from client on the server.
      * Has a timestamp, but if there are some error with concurrency, some events may get lost (this applies to all the collected data (server)).
* Speak of tools in a general matter.

## Test setup

* Discuss different possibilities of test setups.
  + Using console clients
    - Write different for each protocol and framework possibly.
  + Using headless browsers
    - Phantom not supporting WS
    - Slimer does, but is not fully headless and immature.
  + Using real browsers.
    - Limited number of open connections.
* Test machine(s)
* Configuration of servers.
  + Help from Lightstreamer. Mention again later…!

## Choice of setup

* Using real browsers
  + Has its drawbacks
  + Chose Firefox because it actually worked (Chrome refused to connect after six open connections)
  + Play didn’t work with streaming for some reason..
* Describe the application
  + Shared code
  + Reimplemented code (hub and monitor)
  + Framework specific code (like JSONHelper.java)

## Monitoring network traffic

* Wireshark
* Mention some other

## Monitoring of processor

* Could have used some fancy tool.
* Chose resmon..

## Monitoring of memory usage

* Really didn’t need anything fancy here either.

## Limitations

### Meteor

* Meteor has the real time aspect tightly embedded in the core.
* Not feasible for the test case.
* Replaced it with SockJS, which is the library Meteor uses.

### Using browsers

* Max connections
* Overhead?

### 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 scrambled, so there is no way of knowing if the data is correct or not.
* Enter: calculations instead
  + Also has drawbacks.
  + Sources of error.
  + Allowed me to calculate for transports not supported by some frameworks in Firefox. Just to get some comparison, there may be some deviations as to how individual browsers handle different transport mechanisms.

# 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

# WebSockets or HTTP?

# Further work

# 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 TIOBE index ranges programming languages by the number of hits they get on a range of popular search engines. It’s therefore an indication on how widespread a language is. [↑](#footnote-ref-27)