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](#_Toc381131831)

[List of figures vi](#_Toc381131832)

[Preface vii](#_Toc381131833)

[1 The World Wide Web 2](#_Toc381131834)

[1.1 HTTP/1.0 2](#_Toc381131835)

[1.2 HTTP/1.1 3](#_Toc381131836)

[1.3 Problem statement 3](#_Toc381131837)

[1.4 Outline 4](#_Toc381131838)

[1.5 Terminology 4](#_Toc381131839)

[2 Real-time 6](#_Toc381131840)

[2.1 The Real-time Web with HTTP 6](#_Toc381131841)

[2.1.1 Polling 6](#_Toc381131842)

[2.1.2 Long-polling 7](#_Toc381131843)

[2.1.3 HTTP-Streaming 7](#_Toc381131844)

[2.1.4 Comet 7](#_Toc381131845)

[2.1.5 Server-Sent Events 8](#_Toc381131846)

[2.2 WebSockets 8](#_Toc381131847)

[2.2.1 How it works 8](#_Toc381131848)

[2.2.2 The WebSockets API 9](#_Toc381131849)

[2.3 Drawbacks of HTTP techniques 9](#_Toc381131850)

[2.3.1 Really real-time? 9](#_Toc381131851)

[2.3.2 When long-polling becomes polling 10](#_Toc381131852)

[2.3.3 Streaming techniques 10](#_Toc381131853)

[2.4 HTTP was never designed for real-time 11](#_Toc381131854)

[2.4.1 Overhead 11](#_Toc381131855)

[2.4.2 Half-duplex 12](#_Toc381131856)

[2.5 WebSockets is still young 12](#_Toc381131857)

[2.5.1 Know when to use it 13](#_Toc381131858)

[2.5.2 Know how to use it 13](#_Toc381131859)

[2.6 The use of real-time 14](#_Toc381131860)

[2.7 Conclusion 14](#_Toc381131861)

[2.8 Frameworks for real-time web applications 16](#_Toc381131862)

[3 Methodology 18](#_Toc381131863)

[3.1 Selection criteria 18](#_Toc381131864)

[3.1.1 WebSockets support 18](#_Toc381131865)

[3.1.2 Fallback support 18](#_Toc381131866)

[3.1.3 Presentation 18](#_Toc381131867)

[3.1.4 Maturity 18](#_Toc381131868)

[3.2 Description of test application 19](#_Toc381131869)

[3.3 Discussion of use cases 19](#_Toc381131870)

[3.4 Evaluation 20](#_Toc381131871)

[3.4.1 Documentation 20](#_Toc381131872)

[3.4.2 Maintainability 20](#_Toc381131873)

[3.4.3 Simplicity 20](#_Toc381131874)

[3.4.4 Browser support 21](#_Toc381131875)

[3.4.5 Maturity 21](#_Toc381131876)

[3.5 Limitations 21](#_Toc381131877)

[3.5.1 Cloud based solutions 21](#_Toc381131878)

[3.5.2 Rapid development 22](#_Toc381131879)

[3.6 Other choices 22](#_Toc381131880)

[3.6.1 Common UI 22](#_Toc381131881)

[3.6.2 Choice of database engine 22](#_Toc381131882)

[3.7 Selected frameworks 22](#_Toc381131883)

[3.7.1 Socket.IO 22](#_Toc381131884)

[3.7.2 SignalR 23](#_Toc381131885)

[3.7.3 Lightstreamer 24](#_Toc381131886)

[3.7.4 Meteor 25](#_Toc381131887)

[3.7.5 Play Framework 26](#_Toc381131888)

[3.8 Socket.IO – TODO: Rewrite from here 27](#_Toc381131889)

[3.8.1 Why I chose it 27](#_Toc381131890)

[3.8.2 How it works 27](#_Toc381131891)

[3.8.3 Getting started 28](#_Toc381131892)

[3.8.4 Coding environment 28](#_Toc381131893)

[3.8.5 Code structuring 29](#_Toc381131894)

[3.8.6 Serialization 30](#_Toc381131895)

[3.8.7 Maturity 30](#_Toc381131896)

[3.8.8 Documentation 31](#_Toc381131897)

[3.8.9 Implementation of test application 31](#_Toc381131898)

[3.8.10 Summary 32](#_Toc381131899)

[3.9 Lightstreamer 32](#_Toc381131900)

[3.9.1 Why I chose it 33](#_Toc381131901)

[3.9.2 How it works 33](#_Toc381131902)

[3.9.3 Getting started 33](#_Toc381131903)

[3.9.4 Coding environment 34](#_Toc381131904)

[3.9.5 Code structuring 36](#_Toc381131905)

[3.9.6 Serialization 37](#_Toc381131906)

[3.9.7 Maturity 37](#_Toc381131907)

[3.9.8 Documentation 38](#_Toc381131908)

[3.9.9 Implementation of test application 38](#_Toc381131909)

[3.9.10 Summary 39](#_Toc381131910)

[3.10 Play Framework 39](#_Toc381131911)

[3.10.1 Why I chose it 40](#_Toc381131912)

[3.10.2 How it works 40](#_Toc381131913)

[3.10.3 Getting started 40](#_Toc381131914)

[3.10.4 Coding environment 41](#_Toc381131915)

[3.10.5 Code structuring 42](#_Toc381131916)

[3.10.6 Serialization 43](#_Toc381131917)

[3.10.7 Maturity 43](#_Toc381131918)

[3.10.8 Documentation 44](#_Toc381131919)

[3.10.9 Implementation of test application 44](#_Toc381131920)

[3.10.10 Summary 45](#_Toc381131921)

[3.11 SignalR 45](#_Toc381131922)

[3.11.1 Why I chose it 46](#_Toc381131923)

[3.11.2 How it works 46](#_Toc381131924)

[3.11.3 Getting started 46](#_Toc381131925)

[3.11.4 Coding environment 47](#_Toc381131926)

[3.11.5 Code structuring 48](#_Toc381131927)

[3.11.6 Serialization 48](#_Toc381131928)

[3.11.7 Maturity 49](#_Toc381131929)

[3.11.8 Documentation 49](#_Toc381131930)

[3.11.9 Implementation of test application 50](#_Toc381131931)

[3.11.10 Summary 50](#_Toc381131932)

[3.12 Meteor 50](#_Toc381131933)

[3.12.1 Why I chose it 51](#_Toc381131934)

[3.12.2 How it works 51](#_Toc381131935)

[3.12.3 Getting started 52](#_Toc381131936)

[3.12.4 Coding environment 52](#_Toc381131937)

[3.12.5 Code structuring 53](#_Toc381131938)

[3.12.6 Serialization 54](#_Toc381131939)

[3.12.7 Maturity 54](#_Toc381131940)

[3.12.8 Documentation 55](#_Toc381131941)

[3.12.9 Implementation of test application 56](#_Toc381131942)

[3.12.10 Summary 56](#_Toc381131943)

[4 First! **Feil! Bokmerke er ikke definert.**](#_Toc381131944)

[5 Part 1 **Feil! Bokmerke er ikke definert.**](#_Toc381131945)

[6 Part 2 **Feil! Bokmerke er ikke definert.**](#_Toc381131946)

[7 Further work 75](#_Toc381131947)

[8 76](#_Toc381131948)

[9 78](#_Toc381131949)

[10 Appendix 78](#_Toc381131950)

[10.1.1 Choice of server – Part 2 **Feil! Bokmerke er ikke definert.**](#_Toc381131951)

# 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).

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 **Feil! Fant ikke referansekilden.** (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

## SignalR

## Socket.IO

## Atmosphere

## Sails

## Play Framework

## SockJS

## Meteor

## Lightstreamer

## Planet Framework

## Netty

# Servers

## .NET

## Java

## JavaScript

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[[13]](#footnote-13).
* 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[[14]](#footnote-14).
* 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[[15]](#footnote-15).

### 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”[[16]](#footnote-16).

## 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 increasingly popular, and the idea of using JavaScript on the server is very exciting! Over the past couple of years, there has been a dramatic change in the way developers think of JavaScript (TODO: sources). Therefore, it was only natural that I chose at least one framework that uses Node.js as server.

Though there are several modules for Node that provides real-time (TODO: link to node modules), Socket.IO stands out from the crowd. It seems to have the largest community, as it is frequently featured at conferences and generally mentioned many times in traditional forums like Stack Overflow[[17]](#footnote-17).

Furthermore, Socket.IO feels like more than just a Node module. It has its own homepage (TODO: homepage) with some examples and demos–all presented in a good looking and easy to understand fashion. I feel this gives Socket.IO a more professional impression, which makes it stand out even more from some of the other modules that exist that seem more like something someone threw together in a hurry.

Socket.IO doesn’t have a lot of documentation, but what it has gives users a quick overview of the module and how to use it. The API documentation (TODO: docs) uses code samples, which I find more useful than a so-called “wall of text”. There is also a wiki page (TODO: wiki) to give information beyond the API documentation.

Socket.IO strives to blur (TODO: ref homepage) the difference between the different transport mechanisms. WebSockets is the preferred transport, but if the client doesn’t support it, Socket.IO will fall back gracefully[[18]](#footnote-18) to one of the following transports:

* Adobe Flash Socket (TODO: source), which uses Flash to establish a TCP socket connection between the client and the server, thus “mimicking” a WebSocket connection.
* Ajax multipart streaming (TODO: source): An alternative streaming technique to the forever frame technique described in section (TODO: crossref AND should I write this in the essaypart?).
* Forever Frame
* JSONP Polling, which is polling with data type set to JSONP. This allows cross domain requests; something that is not allowed in normal HTTP Polling (TODO: same source as multipart).

### SignalR

First of all, SignalR is one of the few frameworks made specifically for .NET. It is also one of the real time frameworks that has gotten the most attention over the past couple of years[[19]](#footnote-19) amongst real time vendors.

Furthermore, while it is an open source project hosted on GitHub, it has a lot of backing from Microsoft. This is not surprising, as it was originally started by two Microsoft employees that work on the ASP.NET team. This has actually lead to an integration of SignalR in the newest version of Visual Studio (TODO: find source – vimeo vid around 25:00).

SignalR builds on many concepts that should be familiar to .NET developers, like the use of IOC containers for dependency incjection (TODO: source) Nuget for package managing and TODO: one more thing . I believe this, along with the fact that SignalR seems very well built, is why it has gotten a lot of attention very fast.

Something that sets SignalR even more apart from the other frameworks in this thesis, is the abstraction level used to build applications. The use of hubs and *Remote Procedure Calls* (RPC)[[20]](#footnote-20), to do communication between the server and client, thus hiding the event based approach from developers, is very interesting to get a closer look at.

Finally, I sort of “had to” choose SignalR for this thesis. SignalR was actually my first experience with real time technologies back in 2012 and it really sparked my interest. Without this experience, I would probably not have chosen to write my master thesis on this topic.

### Lightstreamer

First of all, Lightstreamer is completely different from all the other frameworks I have found. It is a commercial product from a rather large, European company that has a lot of customers worldwide. Furthermore, no other framework has been on the marked for as long as Lightstreamer, which gives me an unique opportunity to find out how much experience can influence the performance and usability of a real time framework.

As I mentioned, Weswit has a lot of customers worldwide, and if I have interpreted their homepage correctly, Lightstreamer is their only product (TODO: what we do). That means that Lightstreamer is used by a lot of companies in numerous solutions on the web, which is also indicated on the Lightstreamer homepage (TODO: homepage). It is even used by NASA to provide real time telemetry data for the International Space Station (TODO: article from NASA)!

Being a professional product and not “just a GitHub project” give rise to certain demands regarding the documentation of the framework. A well written and explanatory documentation is of course always preferable, but it is even more so for a product people actually pay for. With Lightstreamer, Weswit have obviously had this in mind–there are tons of documentation covering both the server side adapters and the client side APIs.

Another aspect you don’t necessarily get with a non-commercial product is support from the developers or forums. Lightstreamer hosts their own forum on their homepage[[21]](#footnote-21), in which representatives from Weswit frequently answers questions from users. There are also some activity on more general forums like Stack Overflow and Google Groups.

Finally, Lightstreamer, of course (TODO: as of section crossref methodology) supports every major browser. This means that it supports fallbacks to WebSockets, but I have not been able to find out exactly which. Their data sheet is the closest I have found mentioning Comet and HTTP streaming. As I wrote in section (TODO: crossref comet essay), HTTP streaming is covered by the Comet umbrella term, so I’m guessing that they mean some other form of HTTP based real time technology than streaming when they write Comet. A Norwegian blog post suggests that one of the fallbacks is long polling (TODO:allekonsulentene), which makes sense given the separation of HTTP streaming from Comet in Lightstreamer’s docs. Nonetheless, there is fallback support which is what really matters. How it works in practice remains to be seen.

But before I delve into the testing of the framework itself, I need to make some disclaimers. First, I have used the free license of Lightstreamer that does not come with all features included[[22]](#footnote-22). Furthermore, I used the Lightstreamer server itself as application server, which is not the recommended course of action when using Lightstreamer. I chose to do it this way, mostly to keep all the implementations of the test application as similar in architecture as possible. Furthermore, it is generally the case that you would separate the real time aspects of any application to its own server. In this thesis, the purpose is not to implement the whole architecture of a real time enabled web application–just the real time aspects are relevant.

### Meteor

The way Meteor tries to solve web application development is radically different from any other similar framework I have ever seen. While a lot of the abilities, like hot code push, or auto reload, is similar to what other frameworks have (for instance Play framework, see section TODO: code env), it is the total impression it gives that sets it apart.

For instance, it is the first use of Node.js I have seen that truly uses the power of JavaScript on the server by allowing a complete new level of sharing code between the client and the server:

“Write your entire app in pure JavaScript. All the same APIs are available on the client and the server – including database APIs! – so the same code can easily run in either environment.” – (TODO: meteor frontpage)

Another interesting aspect is the fact that real time is the focus of the framework. It will be interesting to look at how this affects performance of the framework, both in typical real time use cases and more static applications.

There are some drawbacks though. Currently, Meteor does not support Windows, which is my development environment for this thesis. Nonetheless, I have chosen to test Meteor using an unofficial fork[[23]](#footnote-23) of the project that brings Meteor to Windows (TODO: win site). Since Meteor runs Node.js underneath it’s hood, I don’t believe that using this unofficial version will cause any issues, but if any problems arises, I will test Meteor in a Linux environment as well.

Another issue is that there are currently only support for MongoDB as database engine. A MySQL project for Meteor does exist (TODO: blog post), but I chose to stay with the database engine provided by Meteor. This because the MySQL support offered by the third party project is not complete, and it changes some of the ways you interact with databases in Meteor. I want to use Meteor as it is supposed to be used (as close as possible at least considering I’m on Windows), and therefore I can settle with knowing that support for other databases are planned in the future of Meteor (TODO: trello board roadmap).

Finally, it should be mentioned that Meteor only received WebSocket support a couple of days before I started using it (TODO: git commit).

### Play Framework

As you might have realized, Play is not a real time framework, so why is it in this thesis? One of the questions I seek to answer is whether or not you need a framework at all to implement scalable real time applications across multiple platforms (browsers) (TODO: crossref problem statement). While Play doesn’t leave you on completely bare ground (TODO: crossref how it works) when it comes to implementing real time, it is quite a lot of steps closer to a manual implementation than what is provided by the other frameworks in this thesis.

Play is one of the few web application frameworks I have seen that promotes an asynchronous application model and real time features. Hence, it stands out from the crowd in this matter. Moreover, it is an increasingly popular framework that is used in production code by some serious actors, like for instance LinkedIn[[24]](#footnote-24).

The asynchronous model that Play offers, means that it should be perfect for implementing real time features. In addition, the framework gives you some helper classes for WebSockets (TODO: ws api doc) and HTTP-streaming (Comet)(TODO: comet api doc) via the forever frame technique (TODO: crossref essay chapter). An interesting byproduct of having these helpers when making real time features, is that it may offer quite a lot of insight in how it would be without the help: If it’s really cumbersome in Play, it is most definitely hard without the help that Play offers. On the other hand, if it’s not too difficult with Play, it means that you can probably write your own helpers by following the Play code (it is open source).

# 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

* Downloading and installing Socket.IO is a task done with one simple command: npm install socket.io.
* Could have had a link to how to install Node.
* Not a lot of documentation, but it is a small library.
  + The structure isn’t all that with some being in the readme and some being in the wiki.
* A little confusing right now as the documentation on GitHub states that it corresponds to the upcoming 1.0 release. A release that has been upcoming for the duration of the work with this thesis.
  + I have not run into problems due to this though.
* Only drawback is that there should maybe have been some larger examples, not only examples with a single html page for instance.

### Simplicity

* Socket.IO is lightweight. In the spirit of Node.
* Strives to be compatible with web application elements like authorization and session management rather than redefining it.
* Quite a lot of options possible, but not needed.
  + Do all with code rather than configuration files.
  + I prefer having files, but this can easily be worked around since it is JavaScript – just use a JSON file and parse it in.
* Offers different channels or “rooms”. Not a requirement, but it is handy in several cases. Can use IGN (Disqus) as an example: Discussion, xxxx people reading this etc..
* Sending data back and forth is a dream.
  + JSON is used behind the scenes which allows you to send an object and receive it in the callback on the other end (TODO: example).
  + A little problem with date, but this is a common JSON problem (TODO: ref).
* In most cases, you don’t need to relate to the concept of a client. Using multiple namespaces is mostly the case where this is necessary.
  + Else, all you relate to is the concept of one or multiple sockets (TODO: example socket.emit, io.sockets.emit).
  + Sending a message, broadcast or not, never requires more than a single line of code.
* Separates different messages by letting you define events. The WebSocket API provides only one message event (onmessage), while Socket.IO allows further separation of this one event. All handled behind the scenes of course.

### Maintainability

* Socket.IO follows the programming principles of Node as well as WebSockets and provides an event based model.
* All code written for Node can be testable, both unit and otherwise.
  + Writing unit tests for the Socket.IO specific code is a little worse. The event driven architecture conceals all logic regarding sending and receiving within callbacks.
  + There are solutions to this of course (TODO: examples (both load and testapp)).
    - Do a call to another module from within each event.
    - Define the callbacks in its own module. This is not the best choice though.
  + To test that some expected code actually executes, one can use integration or functional testing.
  + Another possibility is to inject dependencies to the module with the Socket.IO events and then use the socket.io-client module.
  + None of these things introduce too much overhead compared to “normal” testing.
* A challenge with modules can be to keep them small enough. Larger modules, just as larger classes in OO 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 with many events. Using the namespace construct can help provide a stronger separation of concerns in such cases.

### Browser support

* Supports all major browsers with no quirks.
* Gracefully falls back.
* However, it states that streaming is possible, while in truth, it is not (TODO: source).

### Maturity

* Not 1.0 yet, and has defined 1.0 as “upcoming” for over a year.
* Very varying frequency of commits to the repo. Had a long dead period, but this seems to coincide with the lower lever “engine.io” project.
* All classic signs of a “dead” project.
  + Does not mean that it is immature.
  + On the contrary, Socket.IO feels very stable and mature.
  + Has seen some activity recently as well, indicating that it may be “waking up”.
  + The form of it though seems set, and I have no reason to believe that there will be any major changes in the future (the last was 0.6 to 0.7).
* A lot of projects uses Socket.IO (Trello mentioned before right?), both applications and other frameworks (SailsJS).

## Lightstreamer

## Play Framework

## SignalR

## Meteor

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)

## 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

### 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. [www.knockoutjs.com](http://www.knockoutjs.com) [↑](#footnote-ref-13)
14. For instance a framework for .NET is natural to run using IIS Express. [↑](#footnote-ref-14)
15. WebSockets is the highest level possible, whereas polling is the lowest. [↑](#footnote-ref-15)
16. https://github.com/Atmosphere/atmosphere [↑](#footnote-ref-16)
17. [www.stackoverflow.com](http://www.stackoverflow.com) [↑](#footnote-ref-17)
18. The fallback happens ”behind the scenes” so that developers do not need to worry about it. [↑](#footnote-ref-18)
19. It has been frequently featured at numerous conferences around the world, including NDC (see <http://vimeo.com/ndcoslo/videos>) [↑](#footnote-ref-19)
20. A Remote Procedure Call is when some remote instance, like a browser calls a method on a server (TODO: wiki or something). [↑](#footnote-ref-20)
21. <http://forums.lightstreamer.com/forum.php> [↑](#footnote-ref-21)
22. See full list of features here: <http://www.lightstreamer.com/products> [↑](#footnote-ref-22)
23. 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-23)
24. See Play’s homepage at the bottom: <http://www.playframework.com/> [↑](#footnote-ref-24)