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

Preface

1 Introduction

(First three paragraphs: essay)

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.

1.1 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).

1.2 Outline

1.3 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

2 Background

(Essay)

HTTP, or HyperText Transfer Protocol, is the cornerstone of the World Wide Web. Residing in the application layer of the Internet Protocol Suite (TODO: (3): Internet protocol suite), it provides web pages a mean of linking to other pages–thus creating a "web" of pages.

To enable a web browser to communicate with a server, HTTP uses a request/response pattern (TODO: (4): http 1.1), where the client (browser) makes a request to the server which sends a response back. Underneath this some sort of network layer protocol must be utilized. Most common is the Transmission Control Protocol (*TCP*)(TODO: (5): tcp wiki), but others like (*UDP)* may also be used (TODO: (4): http 1.1).

2.1 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]. 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.

2.2 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

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

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

2.3 Real-time applications

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] 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.

3 Preliminary research

In this chapter, I will look at how real-time has been solved with normal HTTP, and how this compares to WebSockets. Finally, I will make a preliminary conclusion based on the knowledge gained in the work on this chapter.

3.1 The Real-time Web with HTTP

(Essay)

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.

3.1.1 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

[2] 56 vs. 162 pages when copied as they are from http://www.ietf.org into Microsoft Word.
[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.

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).

3.1.2 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.

3.1.3 HTTP-Streaming

HTTP streaming is an old technique introduced by Netscape as early as 1992 - well before even HTTP 1.0 became standard(TODO: (12): A comparison push/pull) Two forms of streaming exist, namely *page streaming* and *service streaming*. The first of the two has the server streaming content in a long-lived TCP-connection. Accomplishing this requires the server to never send the instruction to close the connection - 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] 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.

3.1.4 Comet

Long-Polling and HTTP Streaming are often referred to as Comet or Comet Programming (TODO: (14): Comet: low latency). Comet is an umbrella term that captures different ways to have the server as the initiating part in client/server communication. A rather significant effort has been made to create an official standard for Comet (TODO: (15): Bayeux protocol), but it has yet to become approved by the IETF as a RFC[5]. With the introduction of WebSockets, it may never be.

[4] The forever frame receives JavaScript code wrapped up in script-tags.
[5] Internet Engineering Task Force - Request for Comment series: see http://www.rfc-editor.org/

3.1.5 Server-Sent Events

Let's move on into the borders of Web 2.0 with HTML5s Server-Sent Events (TODO: (16): Html5 server push part 1). Server-Sent Events takes advantage of the "text/event-stream" Content Type of HTML5 (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?

3.2 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 *upgrade* request-header). 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)

3.2.1 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], 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]. 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.

[6] The WebSocket counterparts are ws and wss.
[7] Status code 101

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).

3.2.2 The WebSockets API

As with Server-Sent Events, WebSockets has its own API (TODO: (22): WS API), that provide the *WebSocket* interface. This API is a little simpler than the EventSource interface in my 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)

3.3 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.

3.3.1 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.

3.3.2 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] 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.

3.3.3 Streaming techniques

Using streaming techniques is a different approach than having the client poll for data. With HTTP-streaming and Server-Sent Events, the server is the initiating part 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.

[8] http://www.nbim.no

3.4 HTTP was never designed for real-time

Having introduced the keep-alive flag, chunked encoding and persistent connections in **Feil! Fant ikke referansekilden.** (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.

3.4.1 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] 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].

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.

3.4.2 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.

[9] 87 100 000 bytes * 8 = 696 800 000 bits / 1024^2 = 665 Mbits
[10] 200 000 bytes * 8 = 1 600 000 bits / 1024^2 = 1.526 Mbits

3.5 WebSockets is still young

With new technology comes the almost everlasting issue of backwards compatibility. As mentioned in section (TODO: crossref How it works), the use of the HTTP upgrade request-header ensures this for WebSockets. Implementing it, though, would have been a lot easier if all browsers supported it. As 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] and Socket.io[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.

3.5.1 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.

3.5.2 Know how to use it

An important thing to realize is that WebSockets is not HTTP 2.0. It is a standalone protocol designed to fill the gap of HTTP regarding bidirectional communication. Failing to understand this might cause developers to 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).

[11] http://signalr.net/
[12] http://socket.io/

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.

3.6 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.

3.7 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.

4 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?

This thesis will cover and compare five different frameworks for real time web applications. Frameworks will be selected through a screening process described in section (TODO: crossref). To be able to compare the five frameworks, I need to have a complete impression of each–both what they deliver in form of usability and how they perform. Consequently, the work on(TODO: with?) this thesis will be split into two parts.

In the first part the frameworks will be reviewed from a programmers perspective. This includes aspects like documentation, API, learning curve and other elements concerning general usability.

The second part will look at how well each framework performs. Each framework will be put through a series of load tests for different scenarios. In the end, this will give objective results as opposed to the more opinion-based testing I will do in the first part.

4.1 Selection criteria

Screening for the frameworks that will be featured in the thesis, will be done by the criteria described in this section. Each of the selected framework will have to stand out from the rest in order to be considered for further research.

4.1.1 WebSockets support

A framework does not need to offer WebSockets support in order to be considered, but it must at least mention plans for it, either in a roadmap or somewhere else on the frameworks homepage. However, if WebSockets is not supported, the framework has to offer some unique design or functionality that makes it worthwhile for a deeper study.

4.1.2 Fallbacks

Supporting as many systems as possible is almost always the goal for computer software. For a web application framework that generally means supporting all major browsers. Certain transports are unavailable to older browsers. Therefore, a good real time framework has fallbacks in order to support as many browsers as possible. If a framework has support for one transport only, it is not eligible for this thesis. I will test all frameworks in Google Chrome, Opera, Mozilla Firefox, Internet Explorer 10, 9 and 8. Internet Explorer 7 is still used by many, but fewer and fewer are supporting it. One example is the newest version of the JavaScript framework JQuery (TODO: source).

4.1.3 Documentation

Learning something new without reading about it first is generally a near impossible task. Any framework without documentation will not be considered at all–no matter the impression it gives regarding any of the other criteria in this section. If documentation is present, but incomplete, the framework will need to offer something special to be a part of this thesis.

4.1.4 Presentation

It should be easy to find information about the framework. Furthermore, the information offered should be relevant and not just superfluous text to make it look more appealing. The general impression the framework gives has to be professional, meaning that the homepage, or GitHub page, should not have a lot of flashing lights and other unappealing elements.

4.1.5 Testing

Being able to write automated tests is crucial to make any application maintainable. A framework therefore needs to give some indication that you can write testable code with it. Any framework that clearly states that it does not support unit tests, has to offer some unique design or functionality for it to be considered.

4.1.6 Community

The purpose of any software is to be used by someone. Many real time frameworks are brand new, and thus has very small user bases. This is not necessarily negative, but a very new framework is probably not mature enough to be one of the five frameworks I will study. Older frameworks that still has small communities though, will not be selected.

4.1.7 Cloud based solutions

Cloud based solutions is outside the scope of this thesis. This is mainly of practical reasons, as it is near impossible to compare a cloud based framework with one that runs on a development server locally on my machine (or any other machine). While it would be possible to compare usability from a programmers perspective, performance testing would require sending a lot of data to an external host. If I were to get permission from the manufacturer to do this, it still wouldn't give an even test base when measuring performance.

4.1.8 Other

The following criteria are considered less important, but still count towards the final screening decision:

- **Sessions:** The ability to store session data is not relevant for a library that is meant for direct integration with existing web application frameworks like for instance the .NET Framework. But for others that are meant to run on a stand-alone server, it might be relevant.
- **Tutorials and demos:** Though it is preferable to have tutorials and demos to help with the learning process, it is not required.
- **Collaborators:** If a framework is already in use in production code of well known applications, it is definitely an advantage. However, considering that some frameworks might be quite new, the absence of large collaborators is not considered crucial.

4.2 Evaluation process

The evaluation process will consist of the development of a relatively small application that covers all the common use cases of real time applications: simple messaging and broadcasting of messages–all instant.

4.2.1 Description of application

For each framework, I will implement an auction house, called "Master Auctions". 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.
    - Bids lower than the current bid or bids lower than the minimum price should be disregarded.
- 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].
- MySql will be utilized as database unless it requires substantial workarounds, that may cause the framework to misbehave, to implement it.
- The application will be run locally using either the server bundled with the framework, or a server best applicable for it[14].
- Integration tests will be done by the most applicable way. If there are better ways than using a browser to test (like Selenium[15]), it will be used instead.

[13] www.knockoutjs.com
[14] For instance a framework for .NET is natural to run on the Visual Studio Development Server.
[15] http://docs.seleniumhq.org/

- All tests (integration and unit) will use common testing frameworks in the framework language. For Java: JUnit, for C#: NUnit, for JavaScript: Mocha with some assertion framework like Should.js. TODO: list of technologies

4.2.2 Discussion of use cases

The use cases, registering user and logging in, will test simple one to one communication between the server and the client. The remaining use cases: adding and removing items and placing bids, will test broadcast of one clients action to all other connected clients.

Real time applications also have one other use case: client to client communication (via the server). So called "peer to peer" communication is not part of the specification of the test application simply because it is a subset of simple one to one communication from client to server. The only real difference is that the outgoing message from the server would go to a different client than the origin. Technically this is not worth testing in the test application. It will, however, be a part of the performance test cases (TODO: crossref and write about it).

4.2.3 Common UI

To be able to use a client side view framework like Knockout is getting more and more vital in modern web applications. Such a framework handles a lot of UI updating, and generally makes views more maintainable and easy to write. I feel that is necessary to try to keep the same view as far as it is possible. If the framework under test comes bundled with another view engine however, I will use that instead if it is possible.

4.2.4 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. Using another database for a specific framework is allowed if and only if making it work with MySql requires some workarounds that may change the frameworks original behavior. All hacks that require changing the frameworks source code is also out of the question, and will lead to the usage of one database engine officially supported by the framework.

4.2.5 Choice of server

Running each application locally has both pros and cons. The pros are that I will eliminate potential lag caused by network traffic, and thereby ensuring the experience to be equal to using an external server under optimal conditions. Using a server locally also usually requires little or no configuration, which minimizes the probability of errors due to wrong configuration. On the other hand, running externally would ensure that all available recourses (RAM and CPU) are used solemnly by the server. This does not matter that much to the development of the test application, but it may impact performance tests. I will therefore reconsider server for these (TODO: reconsider).

4.3 Evaluation criteria

When working with each test application, I will do a thorough evaluation of the frameworks from a programmers perspective. Part of this evaluation will be a deeper look into the points from section 4.1, but other aspects will also be evaluated. This section will describe how this process will unfold.

4.3.1 Getting started

While it is not a big part of the developer process, it is still an important factor how easy it is to get up and running with some functioning code. I will emphasis how well the installation process is documented, and whether there are demos or examples or not to help you get started. How steep the learning curve is, will also be discussed under this point.

4.3.2 Coding environment

Does the framework come bundled with an IDE (TODO: forkortelse)? If not, does other, established IDEs support it? Or are you forced to use a basic text editor? Having a good IDE is very useful, especially when working with new technologies. However, it isn't much help if the IntelliSense (TODO: Explain!) support is non-existent. And even more important is debugging opportunities on both client and server. All of these together makes up the coding environment, and I believe having a solid environment is crucial for getting people to use a framework.

4.3.3 Code structuring

Being able to write maintainable code without having to go through a lot of extra work to do so, is even more crucial than having a solid coding environment. This criteria will cover how easy and naturally the application code can be separated into small units[16]. I will also give an evaluation regarding code intrusiveness: Does the framework force developers to apply certain patterns, or is it more free?

4.3.4 Serialization

Passing data back and forth between a client and a server is usually not a straightforward process. Generally, the client is implemented in one language, and the server in another. Data must then be exchanged in a format that can be understood by both, and that's where serialization comes into play. (TODO: figur) Introducing a language that both sides can serialize to and deserialize from, makes data exchange more feasible. As this is a common scenario, I will look at how the frameworks handles this process–if it is handled for you, or if you have to do it yourself manually.

4.3.5 Simplicity

If any part or practice of the framework seems unnecessarily complicated, I will write about it here. Also, if something I expected to be hard is made easy, it will be taken into account under this criteria.

4.3.6 Maturity

A half finished product will most likely never be used in any sort of production code, and is therefore quite useless. With this criteria, I want to evaluate the overall quality of the framework. I also want to give an assessment of how "finished" it felt–how stable and reliable it was during the development process. How well documented the different aspects of the framework is, also counts towards the overall maturity assessment.

4.3.7 Revisited criteria

- **WebSockets support:** If the actual support deviates from the impression the initial screening gave, or if the framework does not support it, I will revisit this criteria.
- **Fallbacks:** If fallback support has to be handled manually, or if it just isn't what was promised, I will need to revise my initial review.

[16] A unit can be either a module, class or just a single file, depending on language.

- **Documentation:** When working with each framework, the documentation will most likely be more actively used. Therefore this criteria will always be revisited. Quality of demos and examples will also be written about under this point.
- **Testing:** Being able to write unit- and integration tests are such important aspects of any application that this also will be revised for each framework.
- **Community:** If I have used, or tried to use the community for help during the developing process, I will revisit this criteria.

4.4 Performance testing

TODO: write about the test cases

5 Development testing

TODO: should I have a sort of "concluding" paragraph under each level 2 subsection?

This chapter will focus on the implementation of the test application described in chapter (TODO: crossref testapp methodology). Each framework has its own subsection where I describe every aspect of the development and thoughts I made during the process. Each section has its own summary, and finally, there is a conclusion giving a nuanced look at what framework solves the different tasks of a real-time application best.

5.1 Socket.IO

Socket.io is a module for Node.js (TODO: nodejs) that provides real-time through pure JavaScript on both server and client. It has been around since 2011 (TODO: last commitpage)[17], and it aims to provide clean and simple real-time across all platforms:

> *"Socket.IOaims to make realtime apps possible in every browser and mobile device, blurring the difference between the different transport mechanisms".*

While it hasn't reached 1.0 yet (TODO: check), it is used in production code by several companies, and it is considered stable. Perhaps one of the most "famous" applications that use Socket.IO is Trello – the online "Scrum Board"[18].

TODO: made by + licence (open source, free)

As mentioned, Socket.IO is a module for Node.js. But what exactly is Node.js? As the name implies, it has to do with JavaScript. More precisely, Node.js is a framework for building network applications using JavaScript. In other words, it is the framework that has made JavaScript enter the realm of backend languages like C# and Java.

Node builds on Google Chrome's JavaScript runtime (TODO: nodejs) and the project has been around for some time. Still, it is not finished and is currently (TODO: write date?) in version 0.10.15.

[17] 0.7 preview was released May 5th 2011
[18] https://trello.com/

However, its Documentation has indicators on each aspect, showing how stable it is, hence making it rather easy to use in a safe way.

5.1.1 Why I chose it

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[19].

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.

As stated in the quote from Socket.IO's homepage in the introduction to this section, Socket.IO strives to blur 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[20] to one of the following transports:

- Adobe Flash Socket (TODO: source), which uses, surprise, 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).

TODO: table, summarize why I chose it?

5.1.2 How it works

TODO!

[19] www.stackoverflow.com

[20] The fallback happens "behind the scenes" so that developers do not need to worry about it.

5.1.3 Getting started

Having Node.js installed on your computer, installing Socket.IO is done via a simple command to the Node Package Manager (*NPM*)[21]. After that you can require it in any JavaScript file in your project (TODO: codelisting).

If you are new to Node.js, the learning curve is somewhat steep. However, this is almost always the case for other frameworks (not just real time frameworks) as well–it is expected that you know how to use the underlying technology.

Still, Socket.IO provides only simple examples, that demonstrates quite simple behavior, on their homepage (TODO: source) and on GitHub (TODO: source). All of these uses just a single HTMLfile, and a single JavaScript file on the server, a case which is quite uncommon in normal web applications. I missed some more information about how to build more complex apps, or at least some more reference to the other frameworks that are used in Socket.IO's examples (like Express (TODO: source)).

5.1.4 Coding environment

As JavaScript code is traditionally just the client part of a web application, it is often written in the same editor as the server code[22]. That may be why most of the examples I could find in videos throughout the web either use a Linux based text editor like Vim or Emacs or the excellent Sublime Text[23], a cross platform editor that has become increasingly popular.

There is also an IDE provided by JetBrains[24] under the name of "WebStorm IDE", which is designed specifically for JavaScript, HTML and CSS. It also has a plugin that allows for Node.js development. This is the environment I chose to use, as it gives good IntelliSense (TODO: explain + crossref appendix), has good syntax highlighting and lets you debug Node.js applications (TODO: figure).

When it comes to debugging, there are a number of options with Node.js applications in addition to WebStorm. Since Node.js is built on Chrome's JavaScript Runtime (TODO: ref Node's homepage), that exposes an extensive debugger over TCP, you can build your own debugger. This is exactly what has been done with the Node Inspector (TODO: ref Git). Using this, you can use Chrome's familiar in-browser debugger to debug your Node.js code (TODO: figure). I actually found this to work better than the WebStorm debugger for certain cases, especially with functions that were used to get back data from a database.

Another option is to use the debugger that comes bundled with Node.js. This is a command line tool that was surprisingly easy and intuitive to use. However, it requires you to write the keyword "debugger", in your code instead of setting breakpoints (TODO: codelist), so it really only works for simple cases, where you don't have to clean up a lot of lines with "debugger" afterwards.

Running a Node.js application is normally done via the command line. You just simply "node" (TODO: cmd list) the main file of your application and navigate to the port it is listening to in the browser. However, when using WebStorm you get the more traditional option of pressing a play button. All

[21] The command "npm install socket.io" installes socket.io and all dependencies right into your project.TODO: codelist instead of footnote?
[22] If the server code is C# for instance, it is common to use Visual Studio also for the client side JavaScript code.
[23] http://www.sublimetext.com/
[24] www.jetbrains.com

output that would normally show in the console, then appears within the IDE's output instead (TODO: figure)–something I found helpful since it meant one less window to toggle between while testing the application manually.

5.1.5 Code structuring

Perhaps the most unfamiliar aspect of JavaScript compared to other languages, is the fact that it is asynchronous (TODO: examplecode)[25]. A common pitfall for JavaScript frameworks is to not fully disclose, either via documentation or function names, whether a function is asynchronous or not. With Socket.IO this is, thankfully, not the case. Socket.IO follows the WebSockets protocol tightly as it provides and event based architecture, and events are always asynchronous. While the WebSockets API only provides a few, standard events (TODO: crossref), Socket.IO lets you used self named events in addition to the standard WebSocket API events.

With traditional, object oriented, languages like Java and C#, code is structured into separate classes, which are normally given their own files. In Node.js, code is structured through the use of modules. Modules is a natural way of separating code within different domains[26], and it is also a very nice way of separating logic that can be used in other applications (TODO: require code list). Furthermore, modules can depend on other modules, which makes you able to build your application using modules as small building blocks. This is how Socket.IO is built up–it is a module, but it depends on a lot of other, smaller modules.

This modularization has both good and not so good aspects. The best aspect of the modularity is diversity: If you need some functionality, for instance a module for communication with a MySQL database, you will almost always not just find it, but find many different alternatives. Using modules also allows for an easy way to contribute to Node.js by making your own modules. This allows for a rather rapid growth of the Node.js project.

However, with so many modules, and seemingly little quality control, developers might end up using quite some time just to find a module that fits their requirements. Furthermore, some modules that are displayed on Node.js's module page (TODO: source), are no longer maintained, which easily can cause problems if you have used it in production code and a bug arises.

Another problem is that modules often tend to favor the use of other modules in their examples and other documentation. This was also the case for Socket.IO, which mostly uses Express. That meant I had to learn more than what I set out to do.

5.1.6 Serialization

When working with JavaScript on the, client the preferred serialization format is, in my opinion, *JSON* (JavaScript Object Notation). With Socket.IO, which uses JavaScript also on the server, I expected serialization to be automatic and abstracted away from me as a developer. Luckily, I was not disappointed, as this is exactly how Socket.IO has solved serialization of objects. When sending data back and forth, one simply sends values, objects or arrays without concern of how they are serialized or deserialized (TODO: codelist). The only problem I had, was when sending Date objects back and

[25] This means that, by default, JavaScript code is non-blocking. You can, however, write synchronous code.
[26] One might for instance put all database logic in one module. If this is a lot of logic, one can split the "main" database module into several, smaller modules, each with a more narrow responsibility.

forth. This was not Socket.IO's, nor Node.js's fault, though, as it has to do with how JavaScript handles deserialization of Date objects[27].

5.1.7 Maturity

Socket.IO is not supported by any large company or otherwise backed by near endless funds. It was started by mainly one person, who is still the largest contributor to the project. It hasn't reached version 1.0 yet, and development seems to be slowing to a crawling pace (TODO: source - git).

Nonetheless, Socket.IO seems very stable and, as I have said, is used in production code (TODO: crossref 1.1). The fact that development has slowed down, may just be a sign that the framework is near complete and very bug free (still, there are quite a lot of issues reported on github(TODO: source)).

However, I must say that I am a little concerned regarding the state of the project. As I said, development seemed to have reach a crawling pace with no new releases recently. The GitHub page states that version 1.0 is just around the corner (the documentation there is even updated to fit this version), but as this is being written, version 1.0 has been "upcoming" for nearly a year.

When it comes to Node.js in general, it is still not a proven platform for large scale applications. Nor is JavaScript itself for that matter. However, building larger applications client side with JavaScript has become more common, but it remains to be seen if the language can make its way onto the server.

Today, I don't think anyone is willing to gamble on choosing Node.js as their primary server solution. However, if some real time functionality is needed, there is no problem using Socket.IO for this, even if the primary server is something else than Node.js. Then, if Node.js doesn't catch on, one can replace it with something else without having to change the entire application.

Delivering real time is something I believe Socket.IO does well, and I must say that I am really excited to see how Node.js will pave the way for JavaScript on the server. If the road ends up full of pot holes or not, remains to be seen.

5.1.8 Documentation

My first impression was that Socket.IO didn't have a lot of documentation, but that what it had was enough. Mostly this is actually true. However, the problem occurs when dealing with the modularity of Node.js. The documentation should do a better job of pointing you in the right direction when it comes to functionality provided by other modules. Because of how it is now, I had to search around a lot and do some trial and error to achieve something as simple as serving the client with multiple files and not just a single "index.html" file as in every single example provided by Socket.IO's documentation (TODO: codelist multiple files and socket.io example).

Node.js itself has a lot of documentation that is well authored. It is also mostly example based, which makes it easier to understand certain aspects. I actually didn't think of the importance of Node.js's documentation before I started working with Socket.IO. But just as it is common to need guidance

[27] When serializing a Date object to JSON, you actually just get the string representation of the object (toString). To deserialize this, one has to initialize a new Date object instance (TODO: js date node soruce).

about core functions of the .NET framework when working with C#, you cannot get by with a Node.js module without getting reference about core features of Node.js itself.

5.1.9 Implementation of test application

As Socket.IO doesn't offer any kind of client side template language, the test application was built using the common UI using Knockout (TODO: as described in.. + ref to git?). The application was build using Node.js version 0.10.8 and Socket.IO version 0.9.11.

To structure my application and harness the asynchronous control flow, I chose a technique that is very common in the asynchronous world, namely promises (TODO: codelist)[28]. The use of promises in JavaScript hasn't been standardized, but most strive to follow one specific proposal (TODO: source). I chose a module for promises called promised-io by Kris Zyp (TODO: source), which closely resembles the client side implementation that JQuery provides (TODO: jquery deferred).

The application's requirements specification (TODO: crossref), states that the application should utilize MySql as database engine. As with everything else in Node.js, I had several different modules to choose from. Felix Geisendörfer's node-mysql was chosen because it had the most comprehensive documentation and it also seemed simple and intuitive to use. No problems arose during development regarding the use of this module, so I still believe that it was the right choice.

There aren't many examples at Socket.IO's homepage regarding how to provide the client with the needed files. That is not within the scope of Socket.IO's functionality either, so it isn't really a big drawback. Nonetheless, I still had to find out how to provide the client side of my application with several files, not just a single HTML file, which is the case in all the examples. Since Socket.IO seemingly recommends Express (TODO: move footnotes?), I chose to use it as well. Express is a web application framework, but for my use case, it was simply used to create the web server of my application (TODO: codelist).

When it comes to browser support, Socket.IO seems to hold what it promises (TODO: crossref). It works fluently in all browsers (TODO: crossref to methodology fallbacks), and it is all handled by Socket.IO behind the scenes.

5.1.10 Summary

Socket.IO is a very solid framework that delivers what it promises (TODO: crossref intro quote). It has seamless fallbacks that enables it to function across all major browsers. Furthermore, it is simple to understand and use, and it is seemingly very stable. The documentation could be a little more comprehensive, and I have some small concerns regarding the pace new releases has come out over the past months. However, this is more likely caused by the fact that Socket.IO is near completion.

TODO: table? Good at, Bad at?

5.2 Lightstreamer

Lightstreamer is a framework made in Java that provides a server specialized for real time applications. The server can communicate with a number of other server technologies via a clever adapter based architecture (TODO: figure). Due to this structuring, you can integrate Lightstreamer with virtually any existing system, as long as the language is supported by it. Currently, Lightstreamer

[28] Also known as deferred or futures.

provides adapters for Java, C# and JavaScript (Node.js). You can even integrate with other languages via the adapter "Remoting Infrastructure" (TODO: doc), though this is actually raw TCP sockets, so it is pretty low level code, and probably a little more work than integrating with a more traditional language adapter.

Client side, Lightstreamer supports numerous different platforms, including a JavaScript API for web applications, several desktop APIs in Java and C#, numerous mobile device APIs and even a generic client that is more low level (TODO: doc).

Behind Lightstreamer is an Italian software company named Weswit, which is actually one of the most experienced companies in the world when it comes to delivering software for making real time applications. Lightstreamer was released as early as 2000 (TODO: ppt about real time), and was one of the first attempts at delivering real time push in web applications without utilizing Java Applets (TODO: wikipedia).

Lightstreamer, unlike the other frameworks in this thesis, is a commercial product. However, they do provide a free license, which is the one I have utilized in my work.

5.2.1 Why I chose it

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[29], 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

[29] http://forums.lightstreamer.com/forum.php

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[30]. 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.

5.2.2 How it works

Lightstreamer incorporates a publish and subscribe model where clients subscribe to items. In this context and item is generally a collection of items (like a database table), for instance can an item be the products in an online auction house.

Using an adapter based architecture makes Lightstreamer easy to incorporate with almost any system. To achieve this, an application using Lightstreamer has to have one metadata adapter and one or more data adapters (TODO: figure or crossref to previous figure).

The metadata adapter's responsibilities include registering subscriptions to users, authenticating and filtering client communication to the appropriate data adapter. In my use case, the metadata adapter mostly routes messages to the data adapter(TODO: codelist).

While the metadata adapter handles incoming communication with users, the data adapter(s) keeps track of the individual items that users can subscribe to. When an item is updated, the data adapter is normally notified via an event listener that handles sending the updates to the clients (TODO: codelist).

On the client, Lightstreamer provides its own grid based template syntax in HTML (TODO: codelist). Furthermore, the framework ships with a JavaScript library for handling these grids and subscriptions. The template engine of Lightstreamer is made to handle received updates on the items the clients subscribes to–there are no mechanisms equivalent to those offered by for instance Knockout, which means that you have to do a lot of UI updates manually via for instance JQuery.

5.2.3 Getting started

Obtaining a free license for Lightstreamer wasn't particularly hard; just fill in a form and you are good to go. Downloading and installing too, was just as you would expect from a well functioning piece of software.

Learning about it and how to use it started out very gently, but it soon proved to be more difficult than what I had hoped. There is only one tutorial for a "Hello World" application that, while it is very

[30] See full list of features here: http://www.lightstreamer.com/products

informative, doesn't really offer too much help when it comes to making something a bit more complex. To help developers move on from the "Hello World" stage, the framework comes bundled with quite a lot of demos. However, these are nearly not documented at all, which can be very frustrating as a lot of the code in the demos handle presentation of data rather than actually sending data back and forth. This was far from clear, and I must say that I spent a bit more time learning how to set up subscriptions (TODO: Cross ref) than I had hoped for.

Another problem is the way the demos are organized. A normal Lightstreamer application would have one metadata adapter and one or more data adapters. Therefore, I expected each demo app to have its own metadata- and data adapter, but when debugging, methods in these were never called. It turned out that all the demo applications share a common metadata adapter class in which all the separate meta data adapters are gathered. A small text on the server start page is the only place where the common meta data adapter is mentioned (TODO: figure), and it was first after I discovered this that the learning process accelerated.

Finally, I missed some indications as to how a database normally would fit into a Lightstreamer application. There is no mention on how to do this in the documentation. In fact, the only mention I found of this was a question in the forum that had been answered, somewhat unclear in my opinion, by a Weswit employee (TODO: source). This may be due to the fact that the Lightstreamer server is supposed to be decoupled from everything except the real time aspects. Therefore, I do not regard this as a drawback with Lightstreamer, but there should perhaps have been some suggestion in the documentation on how a database would fit into the picture.

5.2.4 Coding environment

Giving developers multiple choices regarding programming language, both on the server and client, makes choice of environment very dependent on choice of technologies. Since Lightstreamer is written in Java, the Java adapters are more tightly connected to the framework itself. The .NET version for instance, requires you to build a small proxy server that registers the .NET adapter and handles communication with the Lightstreamer server. In the final paragraph of section (TODO: crossref why I chose it), I argued why I used the Lightstreamer server as an application server, even though this is not the normal use case. By using the .NET adapter, not only would it contradict my intentions, but it could also possibly introduce a bottleneck in my application, since it introduces two more TCP connections to the communication flow (TODO: figure). Therefore, the choice of adapter language fell on Java.

With Java as adapter language, you can choose from all the traditional Java IDEs. I used Eclipse for the server side logic[31], and since the client is pure JavaScript, HTML and CSS, I decided to use WebStorm for this purpose.

Debugging a Lightstreamer application was a little tricky to begin with as there are no documentation on how to attach the debugger to the server. However, a forum post concerning this issue (TODO: forum post) helped me setting it up in a rather simple and straightforward manner. After that, debugging is just like you expect with any Java application.

[31] With server side logic, I mean the adapters.

However, there were some issues. First of all it seems like the source code is either written without any regard to normal naming conventions, or it has undergone some kind of minifying process before it was shipped. A lot of class- and variable-names show up as “t”, “arg0”, “arg1” etc., leaving me as a developer scratching my head from time to time–“What exactly is this parameter really?”. In my opinion, describing names on variables and classes are alpha omega when it comes to writing clean and understandable code.

Additionally, there seems to be some exception handling within the core of the framework that doesn’t get logged in any way. Several times, I would step through my code hunting for where an error had occurred, only to find out that it suddenly stopped working without showing any output at all. After this, the server sometimes shut down, while it sometimes didn’t. Further investigation normally revealed that I had passed a value that was null to somewhere it wasn’t supposed to, which in any other application would result in a red error stack trace in the output window (TODO: figure).

Running the server can be done in two different ways: You can use the provided “Start_LS_as_Application.bat” file, or you can configure Eclipse to launch the server for you. The last option is the only way you can debug the application, so it was the obvious choice. The first option also requires you to manually put related compiled sources into the Lightstreamer folder structure (TODO: figure), which I found rather cumbersome to do while developing the test application.

There is no escape from this when it comes to the client side files though, since the Lightstreamer application server has no other way of serving the client with files that putting them manually into the file system. This problem would, however, go away if you use a separate application server. Therefore, I will not count this aspect towards the final evaluation of Lightstreamer.

5.2.5 Code structuring

Using adapters provides for a very simple separation of concerns. Still, I find a bit odd that you can have several data adapters, but only one meta data adapter. While the meta data adapter handles a lot more than a data adapter, like authentication and such, it could have been useful with more than one of these in a larger application. However, this would require some kind out routing, as it is the meta data adapter that receives messages from the clients, so the way it is now, is really the simplest solution – you can still delegate responsibility to other classes that the meta data adapter administers.

Handling concurrency is always an issue with server applications. Socket.IO is fully asynchronous using events (TODO: crossref), while SignalR’s concurrency issues is handled behind the scenes by the .NET framework (TODO crossref and ref github?). With Lightstreamer you are more in charge of handling concurrency, but you are still urged to use a asynchronous approach using an executor service and event listeners to control the flow in which code gets executed(TODO: codelist – large?). This may be a good thing, as it gives you more freedom in configuring and prioritizing how messages are handled, but it can also be a bit confusing. In my opinion, there should be some default behavior bundled with the framework that you could chose to either hook into or completely override as you see fit. Then you would at least ensure that you can get a basic, well functioning concurrency handling within the application.

Another aspect that has to be handled manually in Lightstreamer is subscriptions. The other "pure" real time frameworks (and also Meteor(TODO: crossref))[32] abstracts this away from the developer, something I find to be very useful as long as there are good mechanisms in place that allows for some form of message routing. With Lightstreamer, you have to keep track of subscriptions via an unique object (a handle), that each subscription receives. If the subscription is "general", in other words something that all users subscribe to, you relate the handle with a listener for that item (TODO: codelist). If you have subscriptions that are user specific, you have to keep track of the subscriptions in a map structure using some unique key to store the handle objects (TODO: codelist).

Client side the code feels somewhat forced. First of all, you must utilize RequireJS[33] to get access to the Lightstreamer client API. Though RequireJS is a very good framework for structuring JavaScript code, it is not the only way, and for some projects it might be a little to comprehensive. In my case, I mixed it with the structure I already had in the common UI, sacrificing a consistent codebase instead of rewriting the whole thing.

As mentioned in section (TODO: how it works), Lightstreamer provides a grid based template syntax for its HTML. For pure adding, removing and updating this works very well, as it allows the server to send a prepared DOM-element that automatically gets added to the grid (TODO: codelist), but if you need more it becomes insufficient rather fast. You are only allowed to data-bind DIV- or SPAN-elements, which effectively restricts any possibility of setting visual traits automatically like you would with Knockout. In order to solve this, I wrote quite a lot of JQuery code that felt rather unnecessary since it did the same as what Knockout handles behind the scenes (TODO: codelist one of the util functions as example).

You can use Knockout if you like, but then you sort of need to "hack" the client side of your application. By using a hidden field and a subscription to a single item, you could send JSON messages back and forth and then process them on the client (TODO: codelist).

However, I believe that Lightstreamer's client is meant to use the template syntax it provides because Lightstreamer is meant to be used for more push based applications. One clear indication of this is the fact that the server has no method, other than the update of an item, to send a message to the client. Simple messaging from server to client is common in more bi directional frameworks like for instance Socket.IO. But, if you keep in mind that Lightstreamer is supposed to coexist with a separate application server, the messaging from server to client could easily be handled by this server instead of Lightstreamer. This would then be through a HTTP response, though, and not via WebSockets.

5.2.6 Serialization

Lightstreamer serializes data that is tightly coupled with the DOM. Server side you send a map containing the fields that are represented in the DOM to a listener. When this arrives at the client, a new DOM-node has been created. When it comes to sending data from the client however, you have to handle serialization your self – both when sending and on the server upon reception. This isn't

[32] A "pure" real time framework is a framework whose main task is to provide real time messaging. Socket.IO, Lightstreamer and SignalR are examples of such frameworks.
[33] http://requirejs.org/

necessarily a bad thing, but it means that, as with other aspects of Lightstreamer (TODO: crossref), you have to write some more logic.

5.2.7 Maturity

Released in 2000, Lightstreamer is, by far, the oldest framework in this thesis. In many ways, you can tell that Lightstreamer is somewhat aging. It is based on the publish/subscribe model, which is quite different from the full bi directional nature of other real time frameworks today. Still, when it comes to pure push, I believe that Lightstreamer is a very stable and mature alternative that delivers high quality to its users.

Nonetheless, there are some flaws. It may actually have become too mature over the years. Compared to for instance Socket.IO and SignalR, Lightstreamer is a huge framework that offers a lot more than just real time technologies with authentication, throttling of bandwidth and other aspects (TODO: datasheet) that normally is handled by an application server. If your application server already does all of this, these extra features of Lightstreamer are redundant.

New versions come out with a little over 2 years in between. The version I used, 5.1.1 came out March 2013 (TODO: downloads page), which means that there are still some time to the 6.0 release. To get up to speed with today's trends, a complete restructuring of the framework might be a good idea. Switching to a more module based framework, they could let developers have more choice in exactly what they want. This would also allow for a better foundation for a more bi directional communication focus rather than the publish/subscribe model they provide today.

But why would they do such a restructuring? Weswit has a large customer base and only good reviews. WebSocekts is a new technology though, and with it comes a new way of solving real time. And while Lightstreamer uses WebSockets, I do believe they would benefit from offering something in addition to the publish/subscribe solution.

5.2.8 Documentation

The documentation is massive–actually it felt a little overwhelming. Still, the API documentation is separated in several documents, one for each API. Something that is almost non-existent though, are examples. With a lot of text and no examples, you get an overview of the concepts, but not how to actually implement them.

This is where Lightstreamer refers you to the demos that comes with the framework. Most of these are rather simple applications, but with a lot of unnecessary code regarding UI formatting. In addition, the code is poorly commented, which is quite strange given the abundant amount of comments in the various configuration files you can use to tweak the Lightstreamer server.

Another rather strange thing with the documentation is that it is very comprehensive, but it does not offer a lot of explanation regarding one of the central concepts of Lightstreamer, namely the different subscription modes. I still don't feel that I have a good understanding of what MERGE, DISTINCT and COMMAND mode really means. The best explanation I found on some of them was actually not in the documentation, but in the forums (TODO: forum post).

5.2.9 Implementation of test application

Lightstreamer allows you to have multiple data adapters for handling several different items. In my case, there is really just one item; the products in the auction house. In addition comes the ability to

register users, log in and view a users current bids. All of these additional features are not something that I believe Lightstreamer would normally handle – they would probably be handled by the application server beforehand.

To circumvent this, I made three different items a client can subscribe to: items (products), logged in users (to keep track over users who are logged in), and bids (per user). The item subscription is initiated the first time the page loads, while the others are initiated on demand.

As described in section (TODO: crossref), Lightstreamer is probably not meant to interact with a database directly. Consequently, there was little help offered, both in the documentation and in the forum, on how to integrate a database directly with Lightstreamer. Hence, solving this was a matter of "best guess", implementing database logic in a logical place within my application logic (TODO: make some kind of figure).

To do the actual interaction, Java applications rely on a MySql connector (TODO: mysql connector). This connector introduces some logic that quickly can become a little messy. In an effort to prevent this, I created a database utility package (TODO: link to git) that I used in the Lightstreamer application, which, in turn, made the database logic very short and readable (TODO: codelist).

5.2.10 Summary

Lightstreamer is a solid framework when it comes to publish/subscribe applications. No other framework in this thesis has been available nearly as long as Lightstreamer, something that shows regarding bi directional communication. However, with the solid platform the framework already has, it has a very good potential when it comes to a restructuring to a module based solution. While Lightstreamer has a large customer base, I believe that it will be unable to compete with the more compact solutions offered by Socket.IO and SignalR in terms of real time applications if they do not do a restructuring similar to what I have described. The reason for this is that, even though Lightstreamer does its job very well, it offers a lot more than what you might generally need in a real time application.

5.3 Play Framework

Play Framework (or just Play) is a *MVC* (Model View Controller) web application framework written in Scala and Java. The project was started by a software developer from the company, Zenexity in 2007, but it was two years later, when it went open source, that it really became popular (TODO: philosophy).

The framework utilizes Akka (TODO: akka) to provide an asynchronous programming model for handling concurrency. Akka is a *"toolkit and runtime for building highly concurrent, distributed, and fault tolerant event-driven applications on the JVM"* – (TODO: akka front page). It is also built using Scala.

Developing a Play application lets you use a lot of different tools and language dialects on both server and client. Server side, you can choose between Java and Scala, whilst on the client, you have a lot of choices. Play uses a template engine to make views, and it comes bundled with one using Scala. However you can easily use any JavaScript based template engine like Knockout, although this requires some use of the Scala template engine as well (TODO: crossref implement or code structure). Furthermore, Play comes bundled with some asset compilers that allow you to use LESS

instead of CSS (TODO: source) and CofeeScript instead of JavaScript (TODO: source), right out of the box.

5.3.1 Why I chose it

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[34].

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).

5.3.2 How it works

In the previous section, I mentioned that Play offers some helper classes for WebSockets and Comet. This does not limit you to use just these means of implementing real time functionality, since Play is, a web application framework with asynchronous capabilities. However, since I used only WebSockets and Comet, I will focus solemnly on those.

Exposing a WebSockets connection with Play is nearly as simple as writing a normal action[35] in a controller. The aforementioned helper class is generic and lets you operate with either JSON or simple strings as message exchange format. To me, the obvious choice was JSON since the client is JavaScript. Initiation a connection is done by simply returning an instance of the helper class (TODO: codelist), rather than returning a result as you would normally do (TODO: or codelist with both?). The handshake is then handled by Play. After the handshake is done, you can use an event based server side architecture to handle incoming messages and disconnections (TODO: codelist).

Using Comet is a little more work, which was what I expected since this requires you to maintain one open connection for pushing data as well as handling incoming POST requests. Nevertheless, it is quite similar to implementing WebSockets, only that the Comet object only provides an outgoing channel. The incoming channel is just normal POST requests. This means that two actions are required in the controller (TODO: codelist). The Comet class also, just like the one for WebSockets, provide an event for handling disconnections.

[34] See Play's homepage at the bottom: http://www.playframework.com/
[35] Controller-methods in a MVC framework are referred to as actions.

As mentioned in section (TODO: play intro), Play uses Akka to provide an asynchronous way of handling concurrency. This is done by using constructs known as “actors” (TODO: wiki). An actor can receive messages, make necessary computations (and even make new actors), and then reply with a proper response. Each actor then, sort of, has its own event loop, forcing all code within the actor itself to be executed with mutual exclusion (TODO: codelist). This all may seem a little complicated, but it is actually quite easy to learn, especially through the excellent documentation of the Akka framework (TODO: source).

5.3.3 Getting started

Getting started with Play was a dream. Almost every aspect of using the framework, from installing to advanced usage, is well documented with accompanying examples. Additionally, there are some samples and tutorials, covering both simple cases as well as more in-depth studies.

However, Play builds on, and uses, a lot of other libraries and frameworks. Although Play does a good job at referencing those external parts’ individual documentations, it still means that the framework itself gets a quite steep learning curve.

While you are mostly helped gently towards the top of that curve, there are some small, but quite unnecessary issues. Some package names in import statements in the tutorials. There were also some links that returned 404 errors in the more advanced tutorial (TODO: zentask + figure?), and in the end, there was some issues that had to be corrected to make it compile.

Relying on quite a lot of different third party libraries and framework, Play can become somewhat vulnerable when some of this third party software becomes outdated. Luckily, Play has a active and alert community that catches such things quickly and solves them. I had some issues when following a tutorial on writing integration tests, which I got help to solve. It turned out that the issue was due to an outdated version of the Selenium Webdriver.

5.3.4 Coding environment

The framework comes bundled with its own console application. This is used to run applications or tests and debugging as well as making new projects. It’s a very functional tool that you don’t really have to worry too much about while developing, since Play uses auto-reload when the application runs in development mode. Due to this feature, your code is recompiled and reloaded in the browser each time you save changes to your project.

Play allows you to use whatever Java or Scala IDE you want (TODO: IDE), and it is even described how to set up projects using the most common IDEs in the documentation. Furthermore, if you don’t like to use an IDE, you can simply use a text editor like Emacs, SublimeText or any other. With the application running from the console, you still get auto-reloading while developing with a text editor.

Debugging a Play application requires you to start the application in debug mode. Then you can attach the debugger from your IDE of choice. While this is a little different than what the other frameworks in this thesis has required, it is actually this method that is the norm. An usual development process has you running the server as a separate process (not started from the IDE). When you do this, you have to attach a debugger to that process in order to debug. For my simple test cases though, it has not been necessary to set up servers in this way, but with Play, this is done automatically through the console, and deploying new changes are handled behind the scenes.

Even though debugging is simple enough most of the time, there were some issues. First of all, while Play comes bundled with Ebean, it does not understand this if you run tests for your models via the IDE–you get an exception. To fix it, you actually have to get Ebean manually (TODO: blogpost) and then reference it through a VM argument in the IDEs run configuration (TODO: figure).

Secondly, debugging the asynchronous nature of Akka is a bit "narrow minded". When debugging code that uses the "await" feature (TODO: codelist), you either have to be very fast, or increase the timeout time beforehand. Neither are preferred ways, but there are no other work around. In my case, this wasn't a big issue, but with a full scale web application built on Play, using actors, it will definitely cause problems. Java obviously has a lot to learn in the terms of asynchronous programming.

5.3.5 Code structuring

Developers who has used ASP.NET MVC will feel very much at home with Plays MVC layout. You may write as many controllers, models or views as you like to keep concerns separated and your code structured–just as one expects from a MVC framework.

There are some slight differences though, especially when it comes to organizing routes. In .NET, routing is, by default, dependent on the names of your controller classes and action names (TODO: MVC blog thing). Play uses a static file, and while you get compilation errors in the browser (TODO: figure), it is not strongly typed in the IDE[36]. In a small application, maintaining the routes file isn't a cause for problems, but in a large application with many routes, I can imagine that it quickly gets out of hand and becomes hard to refactor at a later time.

However, there are probably other means of organizing routes that comes with experience in using Play. When it comes to the models and views part of MVC in Play, you stand very free. Ebean is shipped with Play and used in every example, but you may choose not to use it and write your models however you see fit. The same goes for views: Scala is the default template language, but you may choose not to use it. For .NET developers though, it should feel very familiar as it resembles the Razor view engine that ASP.NET MVC views use (TODO: codelist comparison).

When it comes to concurrency handling, you are, once again, free to use whatever you prefer. Akka, which is most common, gives you a simple, hierarchical way of handling concurrency while maintaining a clean separation of concerns. As I used Play, I learnt more and more about how to use Akka, and looking back at the application structure I ended up with, I see that I probably should have used several actors to handle messaging. Still, one actor does the job, but the code can easily become a little messy with a lot of instance of checks (TODO: codelist).

Any framework needs a simple way of handling dependencies. Maven users are probably accustomed to using a .pom file to list dependencies. With Express for Node.js you have a JSON file (TODO: figure that shows both or two separate..?). Play gives you the Build.scala file, which allows you to list dependencies in a code file in a way that closely resembles what developers already know.

5.3.6 Serialization

As I described in section (TODO: crossref how it works), Play lets you operate with JSON as exchange format. The serialization (TODO: codelist) and deserialization (TODO: codelist) though, you have to

[36] Meaning you don't get IntelliSense while writing new routes.

handle manually. Luckily, the JSON helper class that Play offers by default is the Jackson JsonNode[37], which makes this quite simple. Furthermore, some core features of Play is integrated with Jackson, allowing you to for instance get request bodies as JSON using a simple utility method (TODO: codelist).

5.3.7 Maturity

The framework has been around since 2007, but Play as it is today wasn't introduced before the 2.0 version in 2012 (TODO: wiki). In this release, Scala was incorporated in the core of the framework, instead of being available through external modules in Play 1.x (TODO: philosophy).

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[38], 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 getting started). 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.

In the end though, Play offers a lot, and it tries to let you program in your own way. Whether it is in Scala or Java, using JavaScript or CoffeeScript, LESS or CSS and many more options that are provided right out of the box. Considering that all of this is open source and free, as opposed to ASP.NET MVC, which I see as Play's closest competitor, the overall package isn't all that bad. Still, Play has some way to go though before it reaches the same stability as what .NET offers.

5.3.8 Documentation

Separating the documentation into one section for Java (TODO: source) and one for Scala (TODO: source), allows developers to choose more freely and, at the same time, have no knowledge on the other language what so ever. One can easily imagine a scenario where this separation wasn't present and examples were given as Scala and Java every other time–something that would have made the framework very little appealing very fast.

Another aspect of the API documentation I like a lot is that you can browse previous versions (TODO: screenshot). This means that you are never forced to migrate to the newest version in order to keep up with the documentation. However, there are some drawbacks to this, as it seems that some pages are removed, while the links remain. Even in the newer versions this is an issue.

[37] The version used can be downloaded or viewed here: http://jackson.codehaus.org/
[38] 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.

Relying on a lot of third party frameworks and libraries can soon make you read a lot more than you signed up for. Play does a good job at not describing third party in too much detail, but rather points you to the right external sources if you want to read more.

Finally, the whole documentation is example based, which is, in my opinion, the best way of writing documentation. Ten lines of code says more than a thousand words!

5.3.9 Implementation of test application

Implementing WebSockets and Comet manually with Play was actually quite simple with the utilities that Play offers. As described in section (TODO: how it works), it required three different controller actions and corresponding routes. More interesting is the fact that I managed to share almost all the messaging code done by the single actor in the application. This was accomplished by making two wrapper classes that each inherited from a Socket class containing all common logic. This class only needed one abstract method (TODO: codelist)–the rest is common logic!

Since it seems to be the Play norm, I chose to use the ORM offered by Ebean for my models. A little configuration was required(TODO: codelist), to make Ebean use an external database, but other than that, this proved to be quite straight forward.

Using ORM has left me somewhat divided though. On one hand, it is nice to have all database actions connected to the model they belong to, but on the other, it makes it a little harder to write testable code. Play does offer utilities for this as well (see TODO: cross ref testing), but in the end that doesn't make you code any more structured. A solution that could have been utilized would be to write classes for each model that wraps the database actions, hence making the code a little more loosely coupled. However, I chose to stick to the code style from the tutorials in my application.

Another choice I made was in regard of the use of actors, or more precisely, actor. My application could probably benefit from more than one actor, but the simplest solution was using just one for handling concurrency. Coming in at 193 lines of code (TODO: check), I don't feel that the tradeoff of using several actors is that great.

Providing real time manually means that you have to handle fallbacks client side first and foremost. The server just needs to be able to handle each transport method, but it is the client that has to ask for the right kind. With Plays template engine, it was just a matter of a couple of if-sentences in order to serve the clients with the right JavaScript files(TODO: codelist). The different files initiate the connection, while the rest of the client code just knows it has something called "socket" that can send messages (TODO: codelist – several?).

5.3.10 Summary

Play Framework has some small kinks, but is overall a solid web application framework with an interesting application model (assuming asynchronous). Using it has given me some insight in how it is to implement real time without the support of a framework. With Play, that offers a lot of help in this matter, it wasn't any harder than using a specialized framework.

However, I only implemented one fallback. Furthermore, I just implemented the messaging capabilities that I needed. Real time frameworks often provides the ability to have namespaces, more complex broadcast messaging, more abstraction and generally a simpler application structure. After

using Play to implement real time, my thoughts are that as the complexity of the application increases, the harder it gets to do it all manually, even with Play.

5.4 SignalR

SignalR, or ASP.NET SignalR as it is officially called, is a library written in C# that strives to make implementing real time functionality as simple and straight forward as possible. The project is open source, and its two main contributors are the ones who started it back in 2011 (TODO: github commit page), Damien Edwards and David Fowler, who are both Microsoft employees.

Under the covers SignalR tries to use WebSockets, but if this isn't supported, it will fall back to either Server Sent Events[39], Forever Frame (only Internet Explorer) or long polling. This wide variety of fallbacks means that SignalR is supported in all major browsers. However, it only supports the two newest versions of Chrome, Firefox, Safari and Opera, while IE support stops at IE8.

The library is available through the package manager NuGet[40], and is hosted under the Microsoft.AspNet namespace. More and more libraries and frameworks for ASP.NET is made available in this manner.

In addition to a JavaScript client, SignalR offers clients for Windows Desktop applications, Silverlight, Windows Store and Windows Phone apps, making it a highly usable and cross-platform framework within the C# realm.

5.4.1 Why I chose it

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[41] 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)[42], 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.

---

[39] TODO: SignalR is actually the only framework I've seen that supports Server Sent Events.
[40] http://www.nuget.org/
[41] It has been frequently featured at numerous conferences around the world, including NDC (see http://vimeo.com/ndcoslo/videos)
[42] A Remote Procedure Call is when some remote instance, like a browser calls a method on a server (TODO: wiki or something).

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.

5.4.2 How it works

As mentioned in the previous section, SIgnalR uses an interesting approach for server/client communication. Developers are given a construct called a Hub, in which you can write methods that can be invoked directly from the client. The concept is very similar to MVC in ASP.NET, where you have action methods that corresponds to routes in the application (TODO: codelist), but with some differences.

First of all, there are no need to handle URLs or routing in any other way. The client can simply call a server method (TODO: codelist), and then the framework handles the rest so that the server method is called. Secondly, you can return values as you like, for instance strings, integers or even instances of you own objects back to the client–all in an asynchronous manner (TODO: codelist?).

With the hub, you get a very simple API for handling single messaging and full broadcast. Through the “Caller” and “Clients” objects, SignalR provides some simple methods you can invoke that handles messaging in a correct manner (TODO: codelist).

In addition to the hubs, you can chose to work at a lover lever, using the “Persistent Connections” API. This gives you a more raw implementation of the WebSockets API (still with fallbacks), so that you can tweak the messaging to work the best possible way for your application.

5.4.3 Getting started

Previously, I mentioned that SignalR uses Nuget as package management. This makes installing SignalR into you project just as simple as with Node.js projects: you simply type a command into the package manager console, and then SignalR and all dependencies are loaded (TODO: figure maybe?). There are also additional packages that can be downloaded via Nuget, including a sample package (TODO: wiki package overview or new docs page).

The documentation page (TODO: check) provides some simple examples and some more in-depth tutorials that covers most of the basic to medium advanced aspects of using SignalR. In addition, the SignalR community is very helpful, and the developers are frequently active in the web- and SignalR-based application JabbR, where they happily offer support.

There are some small issues though, mostly regarding aspects of the .NET framework that may be intimidating to inexperienced users. Some of these, advanced things are necessary even in the beginning, and I believe that the SignalR docs will benefit from linking to other resources that cover these aspects. However, for developers with a little experience, these things are a part of their everyday life, so I don’t regard this a big issue.

While working with the test application, I missed an official SignalR forum. Now this is up and running (TODO: forum), giving developers another arena for help and support.

5.4.4 Coding environment

In .NET it's really just one IDE that matters, and that's Visual Studio. Together with the extension, ReSharper[43], you get an excellent coding environment with options for almost any thinkable support.

Normally when developing web applications, you have an external server set up somewhere. I have chosen not to do this (see section TODO). That means that I used Visual Studio's development server to launch my application. This server uses IIS Express, which has some repercussions. I started development with a Windows 7 PC, but IIS Express 8 that I used, does not support WebSockets unless it runs on Windows 8 or Windows Server 2012 (TODO: source). One might think of this as a simple issue as you just need to upgrade in order to fix it. But the reality is that most software companies don't upgrade to the newest version right away–that costs both time and money. Consequently, many SignalR apps may not use WebSockets in the beginning, but luckily, it will start utilizing it as soon as both the server and the client supports it (no configuration needed).

Even though you traditionally use IIS to host .NET applications, SignalR has a way to be easily integrated with existing server farms that doesn't use .NET (TODO: OWIN). Using OWIN[44] (Open Web Interface for .NET) to decouple a web application from the server, you can host a web application (that uses SignalR), outside IIS.

When it comes do debugging, SignalR applications behaves as you would expect from any other .NET application. This means that you can easily attach the debugger to an external process (if using an external server) or launch the development server in debug mode. The same goes for debugging unit-tests and integration tests.

Integration with a database when using ASP.NET usually means connecting to a MSSQL server. In my case, it meant connecting to a MySQL server, a task that I solved using Entity Framework and the MySQL (TODO: write MySQL everywhere!) connector for .NET (TODO: source). This has no impact on the coding style of the application–the code I have can easily be used with a MSSQL server instead.

5.4.5 Code structuring

The way SignalR uses hubs feels very familiar as it keeps in line with standard .NET patterns. It easy to see parallels between controllers in MVC and the hubs. Furthermore, the architecture with hubs and helper objects for handling messaging (TODO: codelist), makes it very easy to focus on the messaging without regarding concurrency and keeping track of clients.

Section (TODO: crossref how it works) mentions that SignalR offers a lower level of abstraction, namely "Persistent Connections". This allows you to expose a SignalR service over HTTP. This gives you more control over your application, but you still have some helper classes to handle messaging. Using this though, you no longer associate with hubs, but with connections. One feature this allows you to implement is the ability to use the connection from wherever you like in your application, thus allowing for instance a normal MVC action method to broadcast real time data to all connected clients or just a group of clients.

In the test application I chose to focus on the use of hubs. This because I believe most usages of SignalR will be with hubs rather than persistent connections as it is the simplest to use. However, in

[43] http://www.jetbrains.com/resharper/
[44] http://owin.org/

chapter (TODO: crossref performance test – use section not chapter reference?) I compare the performance of the two different APIs.

If a hub method returns something (not a void method), you can utilize this on the client. Calling a server method from the client, always returns a JQuery Deferred object, which will be resolved with the return value from the server (if no errors occurred) (TODO: codelist). This lets you program the client using the promise pattern quite a lot.

However, when it comes to broadcasting, there is no other option other than using the "Client" object server side. On the client you then get a pattern that resembles that of Socket.IO quite a lot (TODO: codelist compare). One thing you can do with SignalR though, is have your server method return something to the caller and at the same time broadcast, something that is generally a lot harder with the other frameworks.

5.4.6 Serialization

SignalR handles serialization to and from JSON automatically behind the scenes. This allows developers to pass an object from the client to the server, and also return objects back (TODO: codelist). The objects must match when it comes to field names, but this is standard with all JSON serialization frameworks I have seen. If they don't match, an exception will be thrown, so issues should be easy to fix.

5.4.7 Maturity

SignalR has been around for some time now, but it just recently reached 1.0. However, it seems that updates to the frameworks are coming frequently, and it is being maintained now more than ever. A prerelease of the 2.0 version was up on Nuget in August 2013, so I'm guessing it will be out sometime before 2014.

In addition to having a healthy progression in new releases, SignalR has become very popular very fast within the .NET world. This is probably not that strange considering how many conferences the developers or other members of the .NET team has been to promoting it.

Furthermore, it receives a lot of support from Microsoft, and it is used in the development of Visual Studio and the .NET framework. As I briefly mentioned in section (TODO: crossref why I chose it), Visual Studio 2012 uses SignalR to enable reloading connected applications as you write code–sort of like Play does (TODO: code env, Play), but a little more manual.

There are some issues though, but they do not regard SignalR directly. The fact that WebSockets is a new feature has obvious impacts. To use WebSockets with SignalR, you must run it on IIS8/IIS8 express on Windows 8 or Windows Server 2012 (see section TODO: code env). Seeing how Windows XP is still holding on, over 10 years after its release, it may be some time before software companies have changed all their servers to the newest version.

This is a bit of a shame as it means that it will be hard to utilize the full potential of SignalR performance wise (TODO: crossref performance?) in the next couple of years. However, it might also encourage more software companies to at least change some of their servers to the newest version, in order to use WebSockets.

5.4.8 Documentation

While I was working with the test application (spring 2013), all the documentation was on GitHub (TODO: git docs). But soon after, a new documentation hosted on the official ASP.NET web pages surfaced (TODO: new docs). This is still a work in progress (as of September 2013), but it already contains quite a lot more information than the already somewhat comprehensive GitHub page.

The old documentation was a little lacking in the tutorial section, and it seems like the team was fully aware of this. With the new documentation page, a lot more samples and tutorials have surfaced, with detailed walkthroughs (TODO: getting started – one example) that should enable even complete fresh .NET developers to make something with SignalR.

SignalR has, in a few months, gone from something I regarded as "a Microsoft supported test project" to a fully integrated part of the ASP.NET framework. When I started, GitHub was used for almost every aspect of the documentation. This made it especially hard to find class references, which is very important, especially when it comes to mocking out parts of SignalR in unit tests (TODO: have example in Testing). Now, there is a full featured class reference page for all aspects of SignalR, just like you expect from any .NET library (TODO: ref page), which makes finding reference a breeze.

5.4.9 Implementation of test application

Like, I've mentioned several times now, I chose to use hubs for my test application. Or, more specifically, a single hub since the application isn't really that large. Following standard .NET conventions, I chose to keep most logic outside my hub, and rather have a service that the hub could call into (TODO: codelist). This made coding the server very clean and simple.

Furthermore, I tried to utilize the ability to return a value from the hub methods whenever I just had to do single client communication, like for instance logging in and registering a new user. This allowed me to use a mix of promises and a more event-like approach in my client code–the latter became very similar to Socket.IO (TODO: ref previous codelist).

Since SignalR is a pure messaging framework with no regards to how you implement your JavaScript/HTML client, I chose to use my common UI with Knockout (TODO: crossref methodology?) for this application. Having a good separation of concerns in this code, made it very simple adding the SignalR specific code into it, and the end result was quite similar to the other applications that also uses this UI (TODO: codelist example: Socket.IO vs Play vs SignalR).

5.4.10 Summary

SignalR has established itself as a prominent member of the ASP.NET framework, and it is the best choice by far for all .NET developers when it comes to real time. It is also featured a lot in different conferences, and it has a growing community. Furthermore, SignalR is used in production code in several projects, even in Visual Studio 2012!

The framework makes real time very simple and allows developers to produce applications that would have been a lot more complex without SignalR. It also follows standard .NET conventions, meaning that its very adaptable for any experienced .NET developers.

5.5 Meteor

Meteor is an open source web application framework for writing applications using JavaScript on both the client and the server. To enable this, it runs the server code within a Node.js container. The framework strives to make developing a web application a lot more agile than what it is today, a missing that is in focus throughout their homepage:

> *"This new way should be radically simple. It should make it possible to build a prototype in a day or two, and a real production app in a few weeks."* – (TODO: ref meteor mission)

The framework uses a lot of interesting techniques like hot code push, database everywhere and full real time capabilities right out of the box. All of this has created a lot of talk regarding Meteor–it's a framework that a lot of developers will appreciate if it really can deliver everything it promises.

5.5.1 Why I chose it

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[45] 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).

[45] 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.

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

5.5.2 How it works

Meteor has a rather interesting approach to both web application development as well as real time feature development. Your application will be real time no matter what, which in my opinion is an interesting choice that may cause Meteor to be the wrong choice for certain types of applications. It remains to be seen though how much resources a long lived connection occupies on a server when it's idle (TODO: do performance test and write about it. Crossref).

To handle its real time capabilities, Meteor uses SockJS (TODO: source) – a Node.js competitor to Socket.IO[46]. This is a little more simple framework that simply tires to implement the WebSocket protocol across any browser. Hence, Meteor supports the same browsers as SockJS.

While developing, Meteor does a lot of different magic behind the scenes to make the process go as smoothly as possible. I already mentioned the hot code push functionality, but there's a lot more to Meteor than just updating you application in the browser as you work. Meteor also handles bundling all of the JavaScript files needed by the client as well as building the single HTML-file of the client.

The fact that Meteor is designed to only allow for single page applications is also something that stands out from other frameworks. However, it does not make the development process any harder. On the contrary, Meteor actually builds the single "index.html" file from the templates in your solution. You still need to have an "index.html" file though, and you have to tell Meteor how to build it using a template language (TODO: codelist). The template language bundled with Meteor, and the only one it currently supports officially, is Handlebars[47].

Vowing to provide the same APIs on both server and client (TODO: crossref why I chose it), including database, is something quite strange – how can you possibly access a database on the client? By having the client subscribe to database record sets, Meteor clients actually downloads a copy of the database records, or the records you as a developers chose to publish via Meteors publish and subscribe methods (TODO: codelist).

This has some exciting, but yet disturbing repercussions. First of all, if you try do perform a query to the database you are not allowed to do, you will first see the change before it snaps back when it gets a response from the server (but this can be worked around–see section TODO: code structure) . To me, this sounds like an odd behavior. Luckily, you can keep all your database logic on the server via the use of methods. Using methods with Meteor makes it behave somewhat like SignalR, where the client invokes a method on the server, and gets a value back (TODO: codelist).

5.5.3 Getting started

The unofficial Windows version comes with an installer that handles most of the set up. Setting a PATH variable was the only manual thing I had to do.

[46] SockJS consists of several clients and servers. Meteor, obviously, uses the Node.js server and JavaScript client. SockJS GitHub repo can be viewed here: https://github.com/sockjs/.
[47] http://handlebarsjs.com/

Working with Meteor though, is a quite new way of thinking especially since the framework does a lot of radically different things. Considering this, I feel that the examples provided (TODO: examplepage), really doesn't cover as much as they should. Additionally, the videos that walks through the examples move way too fast in my opinion. They focus on getting you up and running quickly, which is good, but personally, I'd prefer a 45 minute tutorial with coding rather than a 9 minute copy/paste video.

Nevertheless, the documentation has a very thorough explanation of getting started and general concepts. Along with the fact that the samples are very well coded and commented, making them easy to follow. This allows you to have a running app really fast, in spite of all the new and fancy concepts that Meteor introduces.

Once you start to get the concepts, Meteor is actually very intuitive. The command line tool also gives informative error messages that even tells you how to solve certain problems (TODO: uninstall package errormessage). Even the loading of files on the client, which is somewhat complex in meteor (see "Structuring your app" in the meteor docs: TODO: source), becomes natural rather fast.

5.5.4 Coding environment

Meteor, as with Play Framework, comes bundled with a console application that is used for running apps, resetting (database) as well as bundling an application and deploying it to the free servers offered by meteor.com. This bundling feature was somewhat unstable on Windows, but from what I have seen, there are no issues with this on UNIX based systems.

When it comes to IDE support, there was none at the time I wrote my test application. However, WebStorm introduced support for Meteor with WebStorm IDE 7 (TODO: check, if no support, use source that indicates). Without the support, one might still use WebStorm, but that leaves you without IntelliSense on all Meteor specific code. Therefore, I actually found SublimeText, an ordinary texteditor with some support, to be better suited for Meteor development.

Debugging a Meteor app on the server does actually not seem to be officially supported. A quick search for the word debug in the documentation (TODO: source), mostly gives results regarding client side debugging. The one mention on server debug tells you to set an environmental variable that tells Meteor, or more precisely Node.js, to run in debug mode. On Windows however, this does not work, which means that you have to do it a little more manually.

Downloading and installing Meteor, actually downloads the entire source code for the framework. In this code you can find the file that launches the Node.js container that the Meteor server runs within. By modifying this file slightly (TODO: source and codelist), you can enable debug mode on Node.js, allowing you to attach the node-inspector described in section (TODO: crossref Socket.IO). One thing to be aware of though, is that when using this method, the node-inspector shows more than just your code. You also get access to files from the source code. Therefore when debugging, you should only focus on files that reside in the "app"-folder (TODO: screenshot).

Being able to run a large portion of the application on the client will minimize the need for server side debugging. Nevertheless, the need for server code will always be present due to security issues. Therefore, I expect that doing debugging in Meteor will be a lot easier as soon as official Windows support is in place sometime after the 1.0 release (TODO: roadmap).

5.5.5 Code structuring

Meteor uses a fixed way of ordering JavaScript files on the client. As described in the documentation (TODO: source), the order in which files are loaded are predefined by Meteor (TODO: make figure). Moreover, they recommend that you write you apps utilizing packages, which gives more control. Meteor is based on a package structure, so I do believe that for larger apps it will feel natural to use this approach. However, when most examples resolve around using files and not self made packages, many small to medium applications will probably not use packages. To me, it feels a little intrusive that Meteor dictates the ordering, as this is not normal practice with other web application frameworks.

Another peculiar thing is that Meteor seems to encourage the use of global variables–a practice that is generally frowned upon in the JavaScript world (TODO: sources). Global variables are used for collections (database) in every example, which probably leads a lot of users with less JavaScript experience to adapt this way of coding. A better way would be to isolate the collection and rather use a module or package to do read/write operations to it, something you of course can do with Meteor without problems[48].

Other than that, I found structuring my application to be very intuitive and simple. Especially the fact that all code residing outside a folder named "server" will be available on the client. It has become a trend in web application development that more of the application is done front-end rather than on the server (TODO: sources?). Using JavaScript both on the client and the server, Meteor supports this paradigm in a way that no other web application framework I've seen is able to.

Running the server inside a Node.js container allows you to use Node.js modules in your application. At the time I developed my test application though, this was not as simple as it should be. It seems that Meteor wants you to use modules within packages, and not in "normal" file-based applications (see "Writing packages" in the documentation (TODO: docs)). I still managed to do it by using "npm install" within the build folder of my application (TODO: illustrate somehow). This allowed me to access Node.js require in the same way as you do in a smart package (TODO: codelist).

In section (TODO: how it works), I mentioned that Meteor only allows for single page applications. While this is the current trend for web application development, it still feels a little forced. For my simple test case, this was of course no problem (I only use one file), but I imagine that building a large application with Meteor can cause some issues regarding debugging–especially template specific code, as this generally is closely related to the HTML markup.

5.5.6 Serialization

Working with a framework with such a tight relationship between the client and the server, it is expected that serialization is handled behind the scenes. This is the case with Meteor, something that is rather obvious. The shared APIs for accessing a database for instance, would not work if you had to manually serialize objects. This is probably also the reason why MongoDB is the only supported database engine at the time this is being written. I do believe that a SQL integration will

[48] I actually didn't do this, as the focus of this thesis is testing the real time capabilities. Using the global scope shouldn't cause too much decrease in performance in my case.

not be any different to the developers using Meteor, but it will need a lot more work than configuring for MongoDB[49].

5.5.7 Maturity

Meteor has been publicly available since November 2011 (TODO: git commit page), meaning it has been in development for a somewhat long period of time. However, it just recently started to receive appraisal from many developers, boosting the community substantially.

While the framework has come a long way and already offers a rather new way of making web applications. Nonetheless, it still has a long way to go until it reaches a 1.0 release. Furhtermore, in my opinion, the framework will still lack some vital features, like Windows support and SQL integration, even after the 1.0 release (TODO: roadmap).

Potentially, Meteor can become the framework that sets a new standard for how web applications should be developed. If they can deliver every single promise, it will dramatically improve how fast developers are able to deliver production code. In addition, it may introduce a new way of offering real time features, as this is completely integrated with all Meteor applications by default. Still, it remains to be seen whether or not this is a clever approach or not.

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.

Most open source projects have a unstable development cycle. In some periods, there are a lot of activity, and suddenly nothing happens for several weeks or even months. A quick glance at Meteors revision history (TODO: commits) shows that this is not the case for this framework. It is making really good progress, which probably is due to how the development is organized.

Having a steady team of several developers (TODO: meteor/about/people) is partly responsible for this rapid development. More importantly though, is the solid funding the framework received the summer of 2012 (TODO: receives funding). Getting paid normally motivates the continued development of any framework!

5.5.8 Documentation

The fact that Meteor is a work in progress shines through in the documentation. It is not complete, leaving several aspects of the framework just somewhat documented. Furthermore, there are a lot of aspects that have changed in the source code, but that hasn't been completely updated in the documentation. Even worse, some aspects haven't even made it into the documentation yet, which more than a couple of times can lead you to a "trial and error"-based approach to using Meteor.

However, this isn't unnatural given the fact that certain aspects of Meteor can change quickly, meaning that maintaining a documentation that is fully up to date probably would take up too much time. To help you out a little, there are some APIs and features that has text telling you what the

[49] With MongoDB you can store any kinds of objects in the database.

feature will do in version 1.0, which at least gives you a little heads up regarding the state of that particular feature(TODO: screenshot).

Using an example-oriented approach for their documentation, makes it easy to read and learn from. Again though, there may be issues with code examples that are no longer valid, but I did not run into any of such sort. This leaves me to believe that their focus for updating the documentation is to keep all examples up to date.

5.5.9 Implementation of test application

Sharing a lot of code between the client and the server is what really separates Meteor from other frameworks. Therefore, I wanted to take as much advantage of this as possible in my test application. Thus, the server folder of the application does not contain other code than a few methods regarding user operations (TODO: figure or codelist?).

Using a MongoDB database instead of the MySQL database that has been utilized for the other frameworks, meant that I had to do present data a little differently. Especially when it came to the ID (item number) of an item in the database. With MongoDB you cannot have an auto-incrementing integer as ID for a table, simply because MongoDB does not care about types (everything are stored as strings). However, it does have an unique ID for each record, but it is a hash rather than an integer (TODO: figure). For simplicity reasons, I just displayed this hash as the item number in the user interface.

Since Meteor only supports Handlebars officially (TODO: crossref), I chose not to use my Knockout-based common UI. You are free to choose whatever template framework you like with Meteor, but by doing so, you currently cannot access the various helper functions that are built into Meteor (TODO: codelist example).

Out of the box, Meteor offers a package for authentication of users. If I were to develop an application with the framework outside the scope of this thesis, I would definitively used this package as it provides a neat way of handling user accounts with almost no code at all (TODO: show code or videoref). However, the test applications in this thesis need to look similar, something that adding this package would contradict.

5.5.10 Summary

Meteor tries to do something different from many other established web application frameworks. In the process, the developers hope to make the development process a lot more agile, allowing you to go from zero to production in a matter of weeks.

One of the more interesting aspects of Meteor, is the fact that real time the focus on every application. Furthermore, Meteor exposes its database API on the client as well as on the server. These aspects may have repercussions regarding the performance of an application (TODO: test large dataset and write about – summarize here)….

The framework is far from finished, but it has a lot of potential. However, as it appears now, I cannot see how it will be a serious contender for large enterprise applications. Medium to small scale applications though, will probably fit Meteor excellently.

5.6 Conclusion

TODO: here or under 6? 6 under 5? Or?

6 Unit, integration and functional testing

This chapter focuses on testing the various applications' code with automated tests. Testing is alpha omega when building maintainable applications, but sadly this is not taken into consideration by all. In my opinion, testing should be a focus area for any framework right from the beginning. For some of the frameworks in this thesis, that has been the case, but for others it really hasn't. The main focus in this chapter will therefore be on the frameworks that I feel have certain shortcomings when it comes to testing.

6.1 Unit testing

With unit testing there are a few, key challenges that has to be solved. First, what is a unit? Secondly, how to test a unit in isolation?

Lightstreamer, Play and SignalR are all built using object oriented languages (Java, Java/Scala and C#). Unit testing in an object oriented language is traditionally done by testing one class at a time. However, defining what responsibilities one class should hold is another aspect entirely. While working on the Play Framework test application, I followed the programming style of the examples and tutorials from the frameworks homepage (TODO: source). This required the use of Ebean (TODO: source), a framework for object-relational mapping (TODO: wiki or terminology).

With Ebean, each model object, for instance a user object, gets responsibility for doing database interactions. The more standard approach is to keep models that has as little logic as possible and then leave database interactions up to some other class. As any class with database interaction always will have a strong coupling to a datasource, it is a bad subject for unit testing. There are, of course, a lot of ways to circumvent using models with database logic in Play, but I find it a little strange that they lean towards a more strongly coupled way of building applications.

Lightstreamer and SignalR follows a more traditional paradigm, allowing developers to write applications that doesn't require deviations from the programming style of the individual frameworks.

While Play may have some odd quirks in its style, it isn't nearly as odd as the two JavaScript frameworks, Socket.IO and Meteor. I say odd because I was used to the object oriented world before working with these frameworks, so for me it was very interesting to come into a completely different mindset.

JavaScript removes the notion of a class. Instead, one traditionally structures code into modules. A module doesn't necessarily correspond to a class in an object oriented language, but it isn't that far from it. Socket.IO is built using Node.js' modularity, which makes modules a natural way of structuring the application. This makes it easy to write loosely coupled code that is easy to write good unit tests for.

Meteor on the other hand is a completely new way of thinking. While the framework actually uses modules behind the scenes and wraps your own code up in modules, it doesn't feel like it while

coding. Mainly because the way you make functions and objects accessible across files[50] is to add them to the global scope. To me, this blurs the notion of a unit. You still define functions and objects in a file where they naturally belong, but you make them accessible from any other file without having to explicitly inject it. If you're not careful while doing this, you might end up with tightly coupled code that is completely impossible to test, which in turn makes it tiresome to maintain.

Furthermore, this makes the code hard to test on a server. The client code should probably be tested in a client environment like for instance a browser (not necessarily through a functional test). Nonetheless, I chose to test my client code on the server, mostly because with Meteor, all code might run in both environments. The problem is that the server does not have the same notion of the global scope when running in a test scope.

To actually make my files testable, I therefore had to do a lot of workarounds, like adding code to my files to expose the functions under test to Node (TODO: codelist). In my mind, you are not supposed to hack testable code, it should rather be a natural part of the programming process. Mocking out dependencies, was actually very easy through the use of a small module called "unittestling" (TODO: source), that allows you to inject objects into a file that replaces the ones currently there (TODO: codelist).

For now though, Meteor can hide behind being young. Newer versions make use of smart packages (TODO: source), which remind more like modules in the traditional sense. However, it does concern me that Meteor does not have an official testing framework yet.

6.2 Integration testing

In the previous section, I described the somewhat untraditional programming style introduced by Ebean in Play. Looking at all the helper classes and methods provided by Play to enable developers to easily test their Ebean classes, it quickly makes sense why they promote the use of Ebean. Play easily allows you to write tests that runs towards a fake server using an in-memory database (TODO: codelist). This kind of testing is integration testing, but since you use a fake server and a "fake" database, they are closer to actual unit tests. What I think Play tries to achieve is that you don't really need another set of tests as those that run with the in-memory database covers what traditional integration tests aim to check[51].

Lightstreamer, Socket.IO and SignalR are all straightforward, also when it comes to integration testing. Meteor on the other hand is a very interesting case. Considering how tight every aspect of an Meteor app is coupled, integration testing is inevitable. But, with no official testing framework, this is not the simplest of tasks with Meteor right now. There is a framework that can handle this scenario called Laika (TODO: source). Like Meteor, this does not support Windows, but from the examples, it seems to be a viable candidate for an official testing framework for Meteor. It is, at least, something that the developers should incorporate parts of when they actually get around to making one.

Laika also covers the one thing that truly separates Meteor from any other web application framework I have ever encountered: the feature of providing the client with access to the database. However, this part is actually easier to test than the actual integration between application and

[50] Each file gets wrapped up like a module, at least on the client when the app is running.
[51] An integration test traditionally tests your data layer, checking that database connection strings and queries are correct.

database, but it requires the use of a real browser through functional testing as we shall see in the next section.

6.3 Functional testing

All the test applications have the exact same functionality, and while it could have been interesting to write functional tests in each programming language for the individual apps, it would also have been rather repetitive. Therefore, I chose to write a common test suite for all the test applications. Through this test suite, I was able to verify the functionality of each app, including the ability to broadcast data between multiple clients. The code can be reviewed at the thesis' GitHub page (TODO: source).

However, there were some frameworks that also required some more testing. Play actually provides helpers to do functional tests. As with integration tests in Play, you can use an in-memory database when performing functional tests as well, which can be rather useful. It sounds really good, but my experience was that it did not work as it should.

The application utilizes an actor (TODO: crossref or source) to handle concurrency, and the only proper way to test this, is by a functional test. While my common test suite covers this nicely, it would be beneficial to use the method provided by Play itself when making a Play app. Therefore, I tried to get a functional test up and running with Play using their utilities. This did not work due to some problem regarding the actor system. Sadly, I was unable to find the root cause of this problem. There was a bug report on Play's GitHub page (TODO: source), but this was closed before the version I used was released.

Meteor is different from the other frameworks. Since I chose to use the bundled MongoDB (TODO: crossref), the common test suite was not applicable for this application (it depends on the MySql database). Luckily, writing functional tests in JavaScript for Node.js isn't all that different from writing one in Java with Maven. I therefore chose to write a separate test suite for the Meteor application.

7 Load testing

8 Discussion

TODO: Gain of using frameworks?
TODO: When to use real time?
TODO: WS vs HTTP

9 Conclusion