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

# Abstract

Contents

[Abstract i](#_Toc360290225)

[Preface iii](#_Toc360290226)

[1 Introduction 1](#_Toc360290227)

[1.1 Background 1](#_Toc360290228)

[1.2 Essaychapters.. 1](#_Toc360290229)

[1.3 Problem statement 1](#_Toc360290230)

[1.4 Outline 1](#_Toc360290231)

[2 Methodology 1](#_Toc360290232)

[2.1 Selection criteria 1](#_Toc360290233)

[3 Part 1 (change this) 1](#_Toc360290234)

[4 Part 2 (and this) 1](#_Toc360290235)

[5 Results? Or under each part? 1](#_Toc360290236)

[6 Conclusion 1](#_Toc360290237)

# Preface

# 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.

## 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.

## Outline

## Terminology

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

# 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).

## HTTP/1.0

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

At this time, around the mid 90s, CSS too was in its childhood (TODO: (7): Css saga). However, it soon caught people's attention and more and more browsers started to support it (more or less). Embedding a style sheet in a HTML-file adds another object that the client has to download. This is no problem today, but with the HTTP 1.0 protocol it required quite a lot of unnecessary work for both the client and server.

Downloading one element in a HTML-file, or even the HTML-file itself from the server required one TCP request (TODO: figure (2.1)). The server then replied and closed the connection. Getting a HTML-file with a style sheet and three images then required five requests in total, which is obviously inefficient. To circumvent this, some early web applications used several TCP connections at the same time (TODO: (8): Network performance http 1.1). Bear in mind that this was during the old days when download speeds was far from the megabit range.

## HTTP/1.1

Increasing amounts of embedded objects in web pages lead to the creation of HTTP/1.1, which made several vital improvements. One of these was persistent connections. This allowed several request to made over the same TCP connection (TODO: (8): Network performance http 1.1)., and it was a dramatic change at the time, as it gave allowed clients to get several objects in one request.

Another radical improvement was the ability for a browser to cache parts of an object. If the connection to the server was lost half way through the transmission of that particular object, it could later be resumed by using the cached data instead of starting all over. Web applications were also given the possibility of sending chunked data (TODO: (4): http 1.1), letting servers start sending a response without knowing how long it was. In theory, it could be infinite as we shall see in section (TODO: crossref (3.3)).

The authors of the protocol showed great foresight when they made sure that future protocols easily could be made backwards compatible with HTTP 1.1. The *upgrade* request-header (TODO: (9): Key differences) makes it possible for a client to request that another protocol should be used if the server supports it.

Updating from version 1.0 to 1.1 may not seem like a giant leap, but it actually was. Looking at the lengths of the different protocol specifications is an indication of just how much more detailed the 1.1 protocol is[[2]](#footnote-2). Regardless of the advance HTTP 1.1 was, the next step in internet evolution may prove to be even bigger.

## 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]](#footnote-3) and your friends’ activities are displayed to you as soon as it happens (TODO: figure (2.2)).

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

# 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.

## 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.

### Polling

As the very first attempt of providing real-time updates from a server, polling is a fairly simple approach. It works by having the client make normal HTTP-requests, but at a set interval (TODO: (10): Pro Html5). The server then instantly sends back a response - either containing new data or just an empty response if there was nothing to retrieve (TODO: figure 3-1). Polling has obvious flaws like, for instance, how to determine the interval to prevent many empty responses and all the same not flooding the server. Therefore, other mechanisms are far more widespread.

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

### Long-polling

As the name states, Long-Polling is closely related to polling. It basically works the same way, but with one rather important difference. By utilizing the keep-alive header in HTTP 1.1, the connection to the server is kept open after the client has made a response(TODO: (11): Comet and reverse AJAX). This allows the server to send multiple responds over the same TCP-connection (TODO: figure 3-3). If no new data comes to the server in a given amount of time, the connection normally times out (TODO: (12): A comparison push/pull) and the client reconnects through a new HTTP-request.

### HTTP-Streaming

HTTP streaming is an old technique introduced by Netscape as early as 1992 - well before even HTTP 1.0 became standard(TODO: (12): A comparison push/pull) Two forms of streaming exist, namely *page streaming* and *service streaming*. The first of the two has the server streaming content in a long-lived TCP-connection. Accomplishing this requires the server to never send the instruction to close the connection - it remains open throughout the entire course of a client’s session. Service streaming uses a long-lived XMLHttpRequest to send new data, whereas page streaming uses the initial page request. This gives more flexibility regarding the lifetime of the connection.

The most common implementation of this technique today is the so-called forever frame. As mentioned in section (TODO crossref background http1.1), HTTP 1.1 allows a server to send a response without knowing in advance its length. A forever frame is just an iframe that receives script-tags in an everlasting response from a server (TODO: (13): The foreverframe tech) as long as the client is connected, thus using this ability of HTTP 1.1 . Leveraging the fact that a browser executes script-tags[[4]](#footnote-4) whenever it reads them (TODO: (11): Comet and reverse AJAX), the forever frame receives new data from the server wrapped up as such (TODO: figure 3-4). The connection never closes, so each time new data arrives, it is immediately sent to the client and handled appropriately.

### Comet

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

### Server-Sent Events

Let’s move on into the borders of Web 2.0 with HTML5s Server-Sent Events (TODO: (16): Html5 server push part 1). Server-Sent Events takes advantage of the "text/event-stream" Content Type of HTML5 (TODO: (17): Stream updates with..) to push messages to the client without receiving a request first. It is, in other words, a one way communication channel from the server to the client.

Still, the client always has to connect first – “subscribe” to the channel. Then the server can send events whenever new data is available. It can keep the connection open, possibly indefinitely, but at least until it is closed by the client or any intervening proxies. When integrating Server-Sent Events, one can decide how long the connection should stay open and how long it should take before the client reconnects (TODO: (17): Stream updates with..). Server-Sent Events is in other words not too different from long-polling (TODO: figure 3-5).

Unlike long-polling, though, developers using Server-Sent Events have a simple API (TODO: (18): Server Sent Events) that gives access to the *EventSource* interface, which provides straightforward JavaScript code. It allows the server-side to fire events in the browser and, in turn, update the content on the client-side. With the possibility of setting an ID on each message sent, the client can easily reconnect and continue where it left off by having the server look up its ID. This makes Server-Sent Events very robust, but is it powerful enough to match its HTML5 brother, WebSockets?

## WebSockets

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

In December 2011, the WebSockets protocol became a proposed IEFT specification under RFC6455 (TODO: (19): WebSockets becomes). The specification document clearly states that the motivation for WebSockets is HTTPs lack of abilities for bi-directional communication between server and client:

*“The WebSocket Protocol is designed to supersede existing bidirectional communication technologies that use HTTP as a transport layer to benefit from existing infrastructure”* (TODO: (20): WS protocol, section 1.1)

### How it works

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

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

Sending messages back and forth once the connection is up, is a lot more efficient than what HTTP can provide, and it has a lot less overhead too. Header-data in request/response headers in HTTP may accumulate to hundreds of bytes (TODO: (10): Pro Html5), while WebSockets sends messages in frames with only two bytes overhead (TODO: (21): About WS). Frames can be sent both ways at the same time eliminating the need for more than one request at the same time (TODO: figure 4-2).

### The WebSockets API

As with Server-Sent Events, WebSockets has its own API (TODO: (22): WS API), that provide the *WebSocket* interface. This API is a little simpler than the EventSource interface in my mind, having no support for custom events; just for open, close, receiving a message and error.

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

*"Basically it is intended to be as close to just exposing raw TCP to script as possible given the constraints of the Web."* (TODO: (20): WS protocol, section 1.5)

## Drawbacks of HTTP techniques

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

### Really real-time?

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

Thinking real-time, one might want to say that the client should make a new request each time it receives the response of the last. However, this would soon cause any server to crash – unless you have some serious load balancing technology on top, which in turn would lead to a rather expensive solution. Polling the server very often, would also increase the amount of empty responses in cases where data comes to the server in a pulse like manner as shown in (TODO: figure 5-1)on page (TODO: pageref).

How about a longer interval then? Well, with a longer interval, the longer it takes before new data is received, thus making the application less real-time. Even with piggybacking, one cannot achieve anything close to real-time with a longer interval unless the server receives new data at a regular, known interval. As long as this interval isn’t too short, polling may be a good choice for such scenarios. A weather application for instance, might get new updates every hour, which easily can be retrieved by the client using polling.

### When long-polling becomes polling

As I said, long-polling is a lot smarter than polling. Letting the server keep the request open over a longer period of time, ensures that the number of unnecessary requests is a lot less than with polling. Though if the server receives updates at a high rate, the connection will never be able to stay open. Each time the client tries to initiate long-polling, there is always something there waiting for it that makes the server respond immediately (TODO: (10): Pro Html5) – effectively making long-polling work just as regular polling at a short interval. Comparing (TODO: figure 5-1) to (TODO: figure 5-3), one can clearly see that long-polling does not outperform polling as long as the server-side updates are very frequent.

Norges Bank Investment Management[[8]](#footnote-8) provides a counter on their homepage that shows the total value of the Norwegian Government Pension Fund. If each change in that number was a response from the server, it wouldn’t matter if it was polling or long-polling in use – the load on their network would be quite substantial in a short time. This little widget, though, actually fakes real-time as it polls the server every 30 seconds and gets the values from the past 30 seconds.

### Streaming techniques

Using streaming techniques is a different approach than having the client poll for data. With HTTP-streaming and Server-Sent Events, the server is the initiating part rather than the client. One could argue that Server Sent-Events isn’t streaming, but it builds upon some of the same ideas as streaming does with its push approach (even though it can be configured to work more like long-polling – see section (TODO: sse section)).

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

Having a persistent HTTP-connection that sends a lot of data, gives rise to another problem: Proxy-servers and firewalls(TODO: (10): Pro Html5). The nature of the HTTP-protocol may cause these to buffer the response, thus creating a lot of latency for the client (TODO: figure 5-4). Consequently, many Comet-based streaming solutions, like a forever frame, actually fall back to long-polling when buffering is used.

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

## HTTP was never designed for real-time

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

### Overhead

Previously, in section (TODO: crossref How it works), I mentioned that headers in HTTP requests/responses can accumulate to hundreds of bytes (TODO: (10): Pro Html5). In order to get a better picture of why this could be an issue, I will borrow some data from a simple application for comparing polling and WebSockets by Peter Lubbers and Frank Greco (TODO: (23): Benefits of WS). Their simple stock-ticker application polls a server every second to get new data. The counterpart just uses WebSockets to get the same information.

In this particular case, the header-data for the polling application accumulates to a total of 871 bytes. This may not sound like a lot, but when you have clients numbering in hundreds of thousands, the network throughput increases exponentially. A use case with 100 000 users polling every second means that the network in which the server resides, has to deal with 665 megabits per second[[9]](#footnote-9) of throughput. Having the same amount of messages in WebSockets creates only a fraction of that. With 2 bytes of excess data in each frame, it accumulates to a mere 1.5 megabits per second[[10]](#footnote-10).

Using polling to represent HTTP against WebSockets is a little unfair in my opinion, seeing how polling is the naïve approach of achieving real-time. However, it does prove my point: HTTP-headers have much excess data, but most of the time 99% of this data is completely irrelevant for both server and client. Achieving a lot less excess data than this example is possible with HTTP through for example long-polling or Server Sent Events, though nothing will use as little as WebSockets.

### Half-duplex

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

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

## WebSockets is still young

With new technology comes the almost everlasting issue of backwards compatibility. As mentioned in section (TODO: crossref How it works), the use of the HTTP upgrade request-header ensures this for WebSockets. Implementing it, though, would have been a lot easier if all browsers supported it. As this is being written, Internet Explorer has about 14% (TODO: (24): w3Schools) of the browser market with IE8 and IE9 as the most dominant (TODO: (24): w3Schools). None of these supports WebSockets natively, and even though IE10, Chrome, Firefox, Opera and Safari does, it will be several years before developers can safely assume that every single client out there supports WebSockets.

Consequently, applications have to fall back to other, supported techniques when WebSocket support is absent, which in turn leads to more code. Luckily, frameworks like SignalR[[11]](#footnote-11) and Socket.io[[12]](#footnote-12) abstract this away for developers, but sometimes you want more control over the software you create than a framework supplies. And even with frameworks, you might end up having to do some workarounds for certain clients where the fall-back provided by the framework doesn’t suffice.

### Know when to use it

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

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

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

### Know how to use it

An important thing to realize is that WebSockets is not HTTP 2.0. It is a standalone protocol designed to fill the gap of HTTP regarding bidirectional communication. Failing to understand this might cause developers to replace traditional HTTP with WebSockets in applications that don’t really need persistent connections at all. An informative webpage, like Wikipedia, will probably never benefit from using WebSockets. Sure, you get less overhead in request-headers, but on the other hand your application will have to serve mostly idle connections since the only real server to client communication is when the client request a new page (TODO: figure 5-6).

Understanding your application’s environment is another vital aspect. Though WebSockets is supposed to handle proxies and firewalls gracefully (TODO: (10): Pro Html5), you might still encounter some problems – especially if the traffic between your server and the client has to go through an older proxy along the way. Peter Lubbers indicates this in a blog-post from May 2010 (TODO: (25): How Ws interact proxies), and even though this post is rather old, it might be a problem for some. His suggested way of handling the issue is the use of a secure connection (wss:// instead of ws://), which, in my opinion, is a good practice since it makes data encrypted.

## The use of real-time

The World Wide Web has seen many innovations throughout its lifespan, and each time something new comes around, it is hard to determine if it has come to stay. It is always a question of need: Do we really need this? Is it useful to me as a consumer? Real-time is no different from any other new developments; it has to be useful and even to be noticed, it needs to have some form of establishment throughout the web.

There is no doubt that real-time content is very useful in many aspect, and that in others it is even crucial. An auction site with time based auctions completely relies on delivering the latest bid to all users. Forcing their clients to refresh a web page manually to see the latest bid, would render it completely useless. On the other side of the scale we find web sites that utilizes real-time to provide their users with a greater sense of convenience. Getting your friends’ status updates immediately can hardly be seen as crucial, but it does enhance the users’ perception of the experience.

Another interesting development is the increasing amount of real-time content provided by web sites that typically are more static. Most of this has to do with integrating social content like live comment-sections, trending articles and such. Again this is purely to make the content seem more dynamic and make the overall experience better for the users.

Looking at pure web page usage of real-time, it is mostly about the users’ experience. But if we expand our perspective a little, though, it soon becomes clear how much of an impact real-time might have on our lives in the future. Live video streaming is not a strange phenomenon today, but the technology is still in its youth, with buffering issues and broadband capacities as bottlenecks (TODO: figure 5-7). As the technological aspects evolve, I believe we will see a lot more usage of live video streaming across the web. Presumably, WebSockets, with its ability to stream binary data (TODO: (10): Pro Html5),, will play a central part in future improvements to video streams.

## Conclusion

We have seen that even though WebSockets is superior to HTTP when it comes to bidirectional communication, it is not always necessary with a full-duplex channel to achieve real-time content. If most of the communication is from server to client, and the amount of header-data in the HTTP protocol is no cause for problems, it would actually be better to use Server-Sent Events than WebSockets. The need for a fallback for browsers that don’t support this might degrade you to long-polling, which is completely fine as long as the interval in which the server gets updates isn’t too short.

Looking as these aspects leads me to say that HTTP methods may still be a better choice than WebSockets for some real-time purposes. However, if we ignore the need for backwards compatibility, there is no getting away from the fact that WebSockets is superior to HTTP for real-time applications. After all, that was why WebSockets was created in the first place. Nevertheless, HTTP, with Server-Sent Events in particular, remains a strong alternative if you only need real-time push. Long-polling, HTTP-streaming and definitely polling, I think, will be completely outdated in a couple of years – replaced by WebSockets and some Server-Sent Events applications.

I believe that in the future, when current browsers are considered old and WebSockets has been around for a long while, it will be used in most real-time applications. Furthermore, my opinion is that any future versions of HTTP will not incorporate WebSockets – the ywo will remain what they are, namely two separate things.

Social networks like Facebook, collaboration tools like Google Docs and other real-time use cases are already widespread, and that will most likely not change any time soon. Real-time is here to stay, which is good because it provides vast, and yet unseen, possibilities.

Finally, my initial problem was the question of WebSockets’s position in the future of the World Wide Web. Do I believe it is the future? Well, the answer is both yes and no. Yes because it is the future for full-duplex communication applications. It will render HTTP mostly unused for the purpose as soon as the issue of backwards compatibility to clients that don’t support it has vanished. Still, HTTP will remain king of the hill in “traditional” web applications that rely on requesting content in a half-duplex manner.

# 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.

## 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.

TODO: subsections describing each criteria or just a bullet list?

### 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.

### 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.

### 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.

### 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.

### 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.

### 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.

### 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.

### 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.
* **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.

## 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.

### 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]](#footnote-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]](#footnote-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]](#footnote-15)), it will be used instead.
* 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

### 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).

### 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.

### 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.

### 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).

## 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 , but other aspects will also be evaluated. This section will describe how this process will unfold.

### 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.

### 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 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.

### 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]](#footnote-16). I will also give an evaluation regarding code intrusiveness: Does the framework force developers to apply certain patterns, or is it more free?

### 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.

### 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.

### 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.
* **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.

## Performance testing

TODO: write about the test cases

# Part 1 (TODO: change this)

# Part 2 (TODO: change this)

# Results? Or under each part?

# Discussion

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

# Conclusion

1. Embedded objects consisted mostly of images, but also some early forms of style sheets. [↑](#footnote-ref-1)
2. 56 vs. 162 pages when copied as they are from [http://www.ietf.org](http://www.ietf.org/) into Microsoft Word. [↑](#footnote-ref-2)
3. You receive a notification whenever someone likes or comment on an item that is somehow related to your profile (tags, mentioning your name, etc.). See [www.facebook.com](http://www.facebook.com). [↑](#footnote-ref-3)
4. The forever frame receives JavaScript code wrapped up in script-tags. [↑](#footnote-ref-4)
5. Internet Engineering Task Force - Request for Comment series: see http://www.rfc-editor.org/ [↑](#footnote-ref-5)
6. The WebSocket counterparts are ws and wss. [↑](#footnote-ref-6)
7. Status code 101 [↑](#footnote-ref-7)
8. <http://www.nbim.no> [↑](#footnote-ref-8)
9. 87 100 000 bytes \* 8 = 696 800 000 bits / 10242 = 665 Mbits [↑](#footnote-ref-9)
10. 200 000 bytes \* 8 = 1 600 000 bits / 10242 = 1.526 Mbits [↑](#footnote-ref-10)
11. <http://signalr.net/> [↑](#footnote-ref-11)
12. <http://socket.io/> [↑](#footnote-ref-12)
13. [www.knockoutjs.com](http://www.knockoutjs.com) [↑](#footnote-ref-13)
14. For instance a framework for .NET is natural to run on the Visual Studio Development Server. [↑](#footnote-ref-14)
15. <http://docs.seleniumhq.org/> [↑](#footnote-ref-15)
16. A unit can be either a module, class or just a single file, depending on language. [↑](#footnote-ref-16)