Is WebSockets the future of the World Wide Web?

The Real-Time Web



Kristian Johannessen  
1.4.2013

# Abstract

Om kilder: skal man referere til avsnitt?

Contents

[Abstract i](#_Toc346781683)

[1 Introduction 1](#_Toc346781684)

[1.1 HTTP 1.0 1](#_Toc346781685)

[1.2 HTTP 1.1 2](#_Toc346781686)

[2 The Real Time Web with HTTP 2](#_Toc346781687)

[2.1 Polling 2](#_Toc346781688)

[2.2 Long-Polling 3](#_Toc346781689)

[2.3 HTTP Streaming (Forever Frame) 3](#_Toc346781690)

[2.4 Server-Sent Events 3](#_Toc346781691)

[3 WebSockets 4](#_Toc346781692)

[3.1 The protocol 4](#_Toc346781693)

[3.2 The API 5](#_Toc346781694)

[4 Discussion 5](#_Toc346781695)

[4.1 HTTP was never designed for real time web 5](#_Toc346781696)

[5 Conclusion 5](#_Toc346781697)

# Introduction

The World Wide Web has been available to the masses for 20 years [history of the web], and is still considered as 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 has changed the way we used it. Reading an article online today is somewhat different compared to how it was ten years ago. CSS has given HTML documents a better look and feel, AJAX [wiki] made it possible to write web pages with more dynamic content 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. Real time web applications has been around for some time, but previously they have relied on the aging HTTP 1.1 protocol. In this paper, I will investigate the necessity of this new protocol. Does it really improve upon the old fashioned way? If so, is the gain minimal or does it render HTTP obsolete? Answering this requires some digging into the past of the World Wide Web. Furthermore, it demands an understanding of the technologies that WebSockets have been created to improve.(Nytt avsnitt her?) Section 2 gives an overview of the most known alternatives. Section 3 introduces WebSockets and finally, in section 4, I discuss and compare the HTTP-based technologies to WebSocets. But before that, I recommend reading the next two sections, which gives a short introduction to the old HTTP and the version that is used today.

# Background: HTTP

HTTP, or HyperText Transfer Protocol, is the cornerstone of the World Wide Web. Residing in the application layer of the Internet Protocol Suite [har en pdf på det], it provides web pages a mean of linking to other pages - thus creating a "web" of pages.

To enable web browser to communicate with a server, HTTP uses a request/response pattern [rfc2616] 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 [wiki], but others like UDP may also be used [rfc2616]. In this paper the focus will be on TCP. Mainly because of the way WebSocets makes use of one single TCP connection to make full-duplex communication between the server and the client possible (See section...?).

## HTTP/1.0

The 1.0 version of HTTP was created in the World Wide Web's childhood [http 1.0 protocol]. 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, a demand for more lively content arose.

At this time, around the mid 90s, CSS too was in its childhood [css-tingen]. 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 was.

Downloading one element in a HTML-file, or even the HTML-file itself from the server required one TCP request. The server then replied and closed the connection. Getting a HTML-file with a stylesheet 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 [Network performance tingen]. Bear in mind that this was during the old days with dial-up modems - not exactly a 20 megabit internetconnection.

## 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 where persistent connections. Allowing several request to made over the same TCP connection [Network performance tingen], was a dramatic change at the time; giving clients more efficient ways of getting data from servers.

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 was also given the possibility of sending chunked data [rfc2616 3.6.1] letting servers start sending a response without knowing how long it was. In theory, it could be infinite as we shall see in section .

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 [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 protocols specifications is an indication of just how more detailed the 1.1 protocol is. Regardless of the advance HTTP 1.1 was, the next step in internet evolution may prove to be even bigger. I will delve into the world of WebSockets in section .

# The Real-Time Web with HTTP

Recently the concept of real-time web has become a buzzword. Pushing information to the client instantly instead of waiting for the client to request it is how a real-time application works. However, as we have seen, this is not how HTTP works - the client always have 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) up until recently, but with the introduction of WebSockets, all of these may be deprecated. Still, I would like to spend a little time with the old ways before I move on to the future with .

## Polling

As the very first attempt of providing real-time updates from a server, polling is fairly simple minded. It works by having the client doing normal HTTP-request, but at a set interval[HTML boka?]. The server then instantly sends back a response - either containing new data or just an empty response if there was no new data (Figure). 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.

## 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 [Kilde?], the connection to the server is kept open after the client has made a response. This allows the server to send multiple responds over the same TCP-connection (Figure). If no new data comes to the server in a given amount of time, the connection normally times out [kilde!] 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 [A comparison of push and pull tech for ajax]. Two forms of streaming exists, 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 clients session. Service streaming uses a long-lived XMLHttpRequest to send new data, whereas page streaming uses the initial page load. 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 , 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 from a server[the forever frame teq] as long as the client is connected, thus using this ability of HTTP 1.1 . Leveraging the fact that browsers executes script-tags whenever it reads them[Comet reverse ajax], the forever frame receives new data from the server wrapped up as such. 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 [http://infrequently.org/2006/03/comet-low-latency-data-for-the-browser/, og wiki]. 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 [bayeux], but it has yet to become approved by the IETF as a RFC[[2]](#footnote-2). With the introduction of WebSockets, it may never be.

## Server-Sent Events

Moving on into the borders of Web 2.0 with HTML5s Server-Sent Events [api, differences]. Server-Sent Events takes advantage of the "text/event-stream" Content Type of HTML5 [stream updates..] 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.

Through the specification of the API, developers get access to the *EventSource* interface, which provides some easy JavaScript code[api igjen]. 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 of 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 [Begge protokollene] after its successor, was a significant step ahead. However, since the late 90s, no new HTTP protocol has emerged, even though this surely was the authors believes when they made the 1.1 version (see section 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 [ws becomes]The specification document clearly states that the motivation for WebSockets is HTTPs lacking abilities for bi-directional communication between server and client [rfc6455]:

"*The WebSocket Protocol is designed to supersede existing bidirectional communication technologies that use HTTP as a transport layer to benefit from existing infrastructure*" rfc 6455, 1.1

## How it works

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

By having the WebSocket protocol use the same ports as HTTP and HTTPS respectively(80 and 443)[[3]](#footnote-3), the initial handshake can be done via traditional HTTP (Figure). The client states that it wants to use WebSockets, and the server sends a response if it supports it[[4]](#footnote-4). 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 neater 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 [HTML boka side 165], whilst WebSockets sends messages in frames with only two bytes overhead [About WS]. Frames can be sent both ways at the same time eliminating the need for more than one request at the same time.

## The API

As with , WebSockets has its own API [Api fra w3]. Where Server-Sent Events provides the EventSource interface, the WebSockets API 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 rather simple. The simplicity is though 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." Rfc 6455 1.5

# Discussion

Now that we have a general overview of the real-time web and the methods provided by HTTP and WebSockets to enable it, we may go a little deeper and take a more critical stand. Is WebSockets the huge improvement over HTTP, for the purpose of real-time, as it seems to be? Does the HTTP-based techniques perform better at some aspects? And finally; is real-time even important for the future World Wide Web?

## HTTP was never designed for real time web

# Conclusion

1. Images mostly, but also some early forms of stylesheets [↑](#footnote-ref-1)
2. Internet Engineering Task Force - Request for Comment series: see http://www.rfc-editor.org/ [↑](#footnote-ref-2)
3. The WebSocket counterparts are ws and wss. [↑](#footnote-ref-3)
4. Status code 10.1 [↑](#footnote-ref-4)