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 1](#_Toc346781686)

[2 The Real Time Web with HTTP 1](#_Toc346781687)

[2.1 Polling 2](#_Toc346781688)

[2.2 Long-Polling 2](#_Toc346781689)

[2.3 HTTP Streaming (Forever Frame) 2](#_Toc346781690)

[2.4 Server-Sent Events 2](#_Toc346781691)

[3 WebSockets 3](#_Toc346781692)

[3.1 The protocol 3](#_Toc346781693)

[3.2 The API 3](#_Toc346781694)

[4 Discussion 3](#_Toc346781695)

[4.1 HTTP was never designed for real time web 3](#_Toc346781696)

[5 Conclusion 3](#_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 subsections, 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 1.0 kom i internettets barndom
* Nettsider var i stor grad tekstbaserte, eller hadde i det minste ikke mye objekter i seg
  + Grunnet mye bruk av forskere osv
* Flere og flere begynte dog med bilder og annet innhold for å dekorere.
* CSS fantes ikke på denne tiden
* Videre var ikke http 1.0 designet for å håndtere mange objekter i et html dokument[Network performance tingen]
* http 1.0 gjorde én tcp request (med åpning og lukking) pr. objekt
  + Hadde derfor ofte flere TCP connections oppe (på dial up!)
  + Trengte en forbedring

## 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, 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 .

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 .

* Bruker persistent connections
* http 1.1 design goals fra Network performance tingen side 2
* Lar flere objekter hentes ned med en og samme TCP request før den lukkes
* 1.1 har mulighet for å motta deler av et objekt, lagre det i cache og så fortsette der den slapp ved f.eks disconnect.
  + Chuncked encoding (se the forever-frame technique og ref til protokollen)
* Introduserte upgrade request headeren for å gi mulighet til å oppgradere til en tenkt fremtidig oppgradering. Meget fremtidsrettet og smart! Handshake WebSockets [key differences]
* Vise til den artikkelen som indikerer at 1.1 var et relativt stor steg opp.
* Men ikke så stort som neste

Forklare om html 1.0 og 1.1, samt AJAX teknologier som kom i forhold til å hente data. Få frem problemstillingen rundt om Websockets er det nye bra og kule.

Bør fokusere en del på begrensningene i http. Få frem at alle de andre benytter http. Undersøk om du finner noe mer i forhold til store datamengder. Store datamengder er dog ikke aktuelt for real time apper uansett..

# 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 in section .

* Hva vil real time web si?
* Oppdatert innhold til alle klienter med en gang de skjer.
* Steg en: AJAX
  + Gjorde at man kunne oppdatere innhold uten å laste en hel side på nytt
* Applikasjoner som drar nytte av (er nærmest avhenig av) Real Time: Aksjetikkere, auksjonshus, Facebook, Twitter osv..
* Http ble ikke designet for realtime, men det finnes diverse måter å oppnå det (nesten)

Ha figurer som viser data som kommer inn i forhold til forskjellige scenarier. La diskusjonsmateriell utebli herfra - kun informativt! Spørre om real-time i det hele tatt er nødvendig i diskusjonen! Altså flytte det blåe dit.

## 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. 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.

Doing request in a predefined interval like this is a delicate task: What should the interval be? Too short and the server might get flooded with request, too long and the data may be old by the time it reaches the client. The simple answer is that if you don't know how often new data is provided, then polling is generally not a good idea for real-time applications. This is why polling mainly is utilized as a backup solution for the rare occasions it is the only supported mechanism.

* Enkleste form for å hente data. Foregår ved at klienten gjør en request etter ny data. Serveren svarer med en gang enten med ny data eller en tom request [Stream updates with...].
* Kan benyttes til real time, men er oftest siste fallback instans.
* Må gjøres ved et høyt intervall hvis det er ukjent når ny data ankommer.
* Nyttig i applikasjoner hvor man vet i hvilket intervall man får nye data. En værapplikasjon for eksempel.

## Long-Polling

* Ha long-polling og Streaming under COMET? Evt til 2. utkast
* Som polling, men drar nytte av keep-alive muligheten i http 1.1
* Hvis det ikke er noen ny data, holdes connectionen åpen til den
  + Timer ut. Klienten blir da bedt om å koble seg på igjen av serveren
  + Det er data som så sendes over connectionen.
* Bruker mer server ressurser
* Hvis data oppdateres hyppig er det ikke stor forskjell på long-polling og vanlig polling siden serveren aldri får tid til å holde på en request [Html web sockets: quantum].

## HTTP Streaming (Forever Frame)

* Gammelt - Netscape 1992 [A comparison of push and pull tech for ajax]
* Page streaming
  + Streamer innhold i en lengelevende TCP connection.
  + Serveren sender til klienten når en tilstandsendring oppstår.
  + Browseren må oppdatere innhold samtidig som den lytter etter flere endringer.
* Service streaming
  + En XMLHttpRequest som holdes åpen.
  + Mer fleksibilitet i forhold til lengde på connection og hvor ofte nye requests skal gjøres.
* Mest vanlig er å benytte en iFrame for å simulere server push via en åpen TCP connection
* Kalles en forever frame fordi det er en iframe som er uendelig lang grunnet bruk av chuncked encoding (implisitt uendelig).
* Fjerner mye overhead ved å drive å sette opp og ta ned TCP connections hele tiden
* Scripts som sendes til en iFrame utføres umiddelbart.
* Man må håndtere flushing sånn at innholdet ikke blir veldig stort.

## Server-Sent Events

* Beveger oss over i Web 2.0 land - HTML5
* Gir serveren mulighet til å sende data til klienten uten at klienten har gjort en request.
* Benytter Http 1.1.
* Få inn i en diskusjon under konklusjonen eller websockets avsnittet (eller eget?) hvordan SSE kanskje er bedre i de tilfellene du egentlig ikke trenger en toveis kanal.
* Tilbys et eget API gjennom HTML5
* Har automatisk reconnect. Når connectionen er lukket, vil browseren koble seg til igjen etter en viss tid (3 sekunder er standard).
* Man kan også spesifisere at klienten ikke skal koble seg på igjen med Http response 204 [http://dev.w3.org/html5/eventsource]
* Håndterer beskjeder fra serveren med events i browseren. Kan lage egne av disse, noe som gir stor fleksibilitet.
* Kan sette ID på hvert event.
  + Lar browseren benytte Http header "Last-Event-ID" for å koble seg på igjen og fullføre forrige handling [http://www.html5rocks.com/en/tutorials/eventsource/basics/]
* Stor ulempe: proxy'er og brannmurer kan buffere responsen (siden den er så lang), noe som skaper latency [http://www.websocket.org/quantum.html]

# WebSockets

* http 1.1 ferdig i 1999? [sjekk protokoll]
* TCP basert
* Full-duplex på en og samme connection

## The protocol

* Ble laget fordi HTTP ikke er egnet for full-duplex [http://datatracker.ietf.org/doc/rfc6455/?include\_text=1]
  + Mye overhead i HTTP-headere
  + Måtte kanskje benytte flere TCP-connections
  + Tidligere nevnte teknologier holder ikke mål
  + Mottatt data kunne være utadert pga latency allerede ved mottak
* Skal være minimal - ønsker ikke overhead!

"Basically it is intended to

be as close to just exposing raw TCP to script as possible given the

constraints of the Web."

* Designet for å være bakoverkompatibel - dyktig design av http 1.1 med upgrade header!
* Handshake
  + Hente figurer
  + Klienten ber om å oppgradere til WS
  + Hvis serveren sier ja, bryter HTTP connectionen ned og WS tar over på samme TCP/IP connection
* Benytter samme porter som HTTP og HTTPS (80 og 443)
* Kan sende både binærdata og tekst
* Skaper er "tunnel" gjennom brannmurer og proxier sånn at data ikke blir bufferet [ws.org]
* Frames
  + Kan sendes full-duplex begge veier samtidig
  + Har bare 2 bytes med kontrolldata i motsetning til HTTP-resonser som kan være flere 100 [http://www.websocket.org/quantum.html]
* Definerer en ny type URI: ws:// og wss://

## The API

* Slik som SSE er det definert et eget API
* Har diverse events og metoder for enkel tilkobling og frakobling
* Ser ikke ut til å støtte custom events på samme måte som SSE

# Discussion

Blabla bla

## HTTP was never designed for real time web

# Conclusion

1. Images mostly, but also some early forms of stylesheets [↑](#footnote-ref-1)