**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](#_Toc347773790)

[List of figures iii](#_Toc347773791)

[1 Introduction 1](#_Toc347773792)

[2 Background: HTTP 1](#_Toc347773793)

[2.1 HTTP/1.0 2](#_Toc347773794)

[2.2 HTTP/1.1 2](#_Toc347773795)

[3 The Real-Time Web with HTTP 3](#_Toc347773796)

[3.1 Polling 3](#_Toc347773797)

[3.2 Long-polling 4](#_Toc347773798)

[3.3 HTTP Streaming 4](#_Toc347773799)

[3.4 Comet 5](#_Toc347773800)

[3.5 Server-Sent Events 5](#_Toc347773801)

[4 WebSockets 6](#_Toc347773802)

[4.1 How it works 6](#_Toc347773803)

[4.2 The API 8](#_Toc347773804)

[5 Discussion 8](#_Toc347773805)

[5.1 Drawbacks of HTTP techniques 8](#_Toc347773806)

[5.1.1 Really real time? 8](#_Toc347773807)

[5.1.2 When long-polling becomes polling 9](#_Toc347773808)

[5.1.3 Streaming techniques 10](#_Toc347773809)

[5.2 HTTP was never designed for real-time 11](#_Toc347773810)

[5.2.1 Overhead 11](#_Toc347773811)

[5.2.2 Half-duplex 11](#_Toc347773812)

[5.3 WebSockets is still young 12](#_Toc347773813)

[5.3.1 Know when to use it 12](#_Toc347773814)

[5.3.2 Know how to use it 13](#_Toc347773815)

[5.4 The use of real-time 13](#_Toc347773816)

[6 Conclusion 14](#_Toc347773817)

[References 16](#_Toc347773818)

# List of figures

[Figure 2‑1: Several request to get the whole page. 2](#_Toc347773351)

[Figure 3‑1: Normal polling to the left, polling with piggybacking to the right 3](#_Toc347773352)

[Figure 3‑2: Long-polling 4](#_Toc347773353)

[Figure 3‑3: A client using the forever frame technique receives script-tags from the server. 5](#_Toc347773354)

[Figure 3‑4: Server-Sent Events resembles long-polling. 6](#_Toc347773355)

[Figure 4‑1: WebSocket handshake 7](#_Toc347773356)

[Figure 4‑2: Frames can pass both way through one connection. 7](#_Toc347773357)

[Figure 5‑1: Short poll interval gets new data fast, but demands server resources 9](#_Toc347773358)

[Figure 5‑2: Long interval makes the client miss a lot of updates. 9](#_Toc347773359)

[Figure 5‑3: Long-polling has no benefit over polling at high update-rates. 9](#_Toc347773360)

[Figure 5‑4: Intervening proxy buffering response-stream from server. 10](#_Toc347773361)

[Figure 5‑5: Two TCP connections simulating full-duplex. 12](#_Toc347773362)

[Figure 5‑6: No need for exchanging messages leaves idle connections. 13](#_Toc347773363)

[Figure 5‑7: Hopefully, this scenario will be a thing of the past in a not too distant future. 14](#_Toc347773364)

# Introduction

The World Wide Web has been available to the masses for 20 years [1] [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 [2] [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. Section gives an overview of the most known alternatives. Section introduces WebSockets and finally, in section , 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(more on that in 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 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 [Network performance tingen]. Bear in mind that this was during the old days with dial-up modems - not exactly a 20 megabit internet connection.



Figure ‑: Several request to get the whole page.

## 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 WebSockets.

## 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 nothing to retrieve (). 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. piggyback

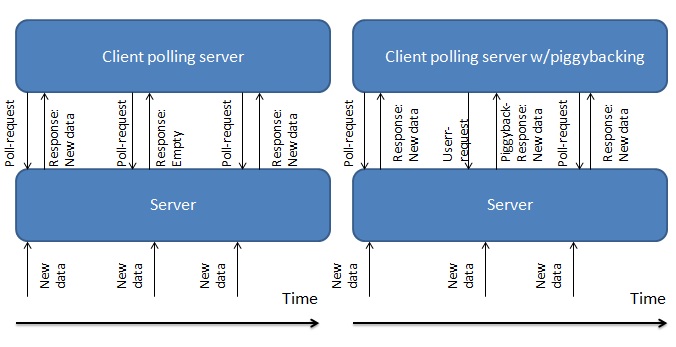


Figure ‑: Normal polling to the left, polling with piggybacking to the right

There is a way to improve a little upon polling, namely piggybacking [3] [Comet reverse ajax]. Polling the server at a regular interval is usually done in parallel to other HTTP-requests initiated by client actions. These actions, of course, also gets 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 ().

## 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 (). 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.



Figure ‑: Long-polling

## 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 [3] [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.

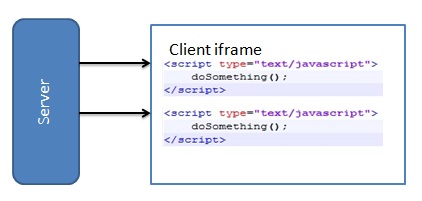


Figure ‑: A client using the forever frame technique receives script-tags from the server.

## 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

Let’s move on into the borders of Web 2.0 with HTML5s Server-Sent Events [today java.net tingen, 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.

Still, the client always has to connect first – “subscribe” to the channel. Then the server can send events whenever new data is available. It keeps 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 [api? Stream?]. Server-Sent Events is in other words not too different from long-polling ().

Unlike with long-polling though, developers using Server-Sent Events has a simple API [api] that gives access to the *EventSource* interface, which provides some easy 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 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?



Figure ‑: Server-Sent Events resembles long-polling.

# 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 section ), 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 (). 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.



Figure ‑: WebSocket handshake

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 ().



Figure ‑: Frames can pass both way through one connection.

## The API

As with Server-Sent Events, 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. Can any of the techniques that uses HTTP challenge the need for WebSockets? Do they surpass WebSockets in at least some aspects? What about the protocols themselves – does WebSockets seem so amazing just because HTTP hasn’t realized its full potential? And finally; is real-time even that necessary for the future of The World Wide Web?

## Drawbacks of HTTP techniques

In section , 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, and there are several reasons why. 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 on page 9.



Figure ‑: Short poll interval gets new data fast, but demands server resources

How about a longer interval then? Well, the longer the interval, the less real-time the application gets (). Even with one cannot achieve anything close to real-time with a longer interval unless the server receives new data at a regular, known interval. A weather application for instance, might get new updates every hour, which easily can be retrieved by the client using polling.



Figure ‑: Long interval makes the client miss a lot of updates.

### 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 are a lot less than with polling. Though if the server gets updates at a high rate, then 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 [html boka] – effectively making long-polling work just as regular polling at a short interval. Comparing to , one can clearly see that long-polling does not outperform polling as long as the server-side updates are very frequent.



Figure ‑: Long-polling has no benefit over polling at high update-rates.

Norges Bank Investment Management[[5]](#footnote-5) 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.

### 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 ).

Since the (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 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 [html5 boka]. The nature of the HTTP-protocol may cause these to buffer the response, thus creating a lot of latency for the client (). Consequently, many Comet-based streaming solutions, like a forever frame, actually falls back to long-polling when buffering is detected.



Figure ‑: Intervening proxy buffering response-stream from server.

A forever frame makes the developer write some additional code to handle the incoming scripts. With the EventSource interface of Server-Sent Events, developers has a more powerful toolbox for wrapping up the incoming events (see section ). Utilizing pure eventhandlers also ensures that there is no need for cleaning up the incoming events – they 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 ), 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 over WebSockets: its design and, simply, its age.

### Overhead

Previously, in section I mentioned that headers in HTTP requests/responses can accumulate to . 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 [quantum]. 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[[6]](#footnote-6) 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 1.526 mega bits per second[[7]](#footnote-7).

Using polling to represent HTTP up 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 has much excess data in them, 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

Finished in the 90s and 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 solemnly for the purpose of full-duplex [ws protocol tipper jeg går] 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 has to use several TCP-connections (). 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 very beginning of this paper (see section ) 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.

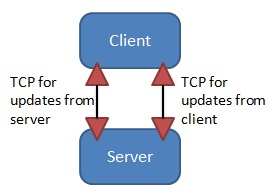


Figure ‑: Two TCP connections simulating full-duplex.

## WebSockets is still young

With new technology comes the almost everlasting issue of backwards compatibility. As mentioned in section , the use of the HTTP upgrade request-header ensures this for WebSockets. Implementing it though is another story. As this is being written, Internet Explorer has about 14% of the browser market with IE8 and IE9 as the most dominant[w3 org]. None of these supports WebSockets natively, and even though IE10 does, it will be several years before developers can safely assume that every single client out there supports WebSockets.

Consequently, applications has to fall back to other, supported techniques when WebSocket support is absent, which in turn leads to more code. Luckily, frameworks like SignalR[[8]](#footnote-8) and Node.js[[9]](#footnote-9) abstract this away for most people, 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 real-time comment-section and a chat room respectively.

Leveraging WebSockets for the first example would work excellent, 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 across 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 those who 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 never benefit from using WebSockets. Sure, you get some 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 ().



Figure ‑: No need for exchanging messages leaves idle connections.

Understanding your applications environment is another vital aspect. Though WebSockets is supposed to handle proxies and firewalls gracefully [html boka], 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 [kilden], 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 just healthy 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 than any other new developments; it has to be useful and to even be noticed, it needs to have some form of establishment throughout the web.

It 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 latest bid, would render it somewhat useless. On the other side of the scale we find web sites that utilizes real-time to provide their users 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 is 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 a practicality. 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 (). As the technological aspects evolve, I believe we will see a lot more usages of live video stream across the web. Presumably, WebSockets, with its ability to stream binary data [html boka] will play a central part on future improvements to video streams.

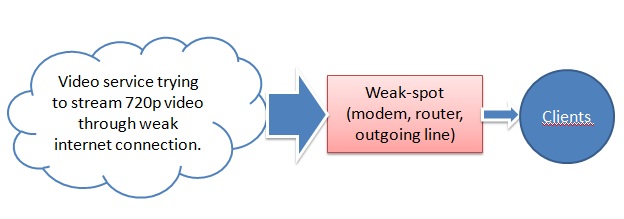


Figure ‑: Hopefully, this scenario will be a thing of the past in a not too distant future.

# 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 browser that don’t support this though might degrade you to long-polling, which is completely fine as long as the push-interval 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 for the sake of real-time in a couple of years – replaced with 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 – they 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 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 useless 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 relies on requesting content in a half-duplex manner. Even push applications will probably mostly use HTTP. Mainly this is because you don’t need to overuse open connections. A connection that is idle 90 % of the time (or even more) is a waste of server resources, and with WebSockets that scenario would be inevitable in a purely informative application like for instance Wikipedia.

# References

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 101 [↑](#footnote-ref-4)
5. <http://www.nbim.no> [↑](#footnote-ref-5)
6. 87 100 000 bytes \* 8 = 696 800 000 bits / 10242 = 665 mbits [↑](#footnote-ref-6)
7. 200 000 bytes \* 8 = 1 600 000 bits / 10242 = 1.526 mbits [↑](#footnote-ref-7)
8. <http://signalr.net/> [↑](#footnote-ref-8)
9. <http://nodejs.org/> through the use of extension modules like socket.io: <http://socket.io/> [↑](#footnote-ref-9)