HTML5 Web Sockets:A Quantum Leap in Scalability for the Web

Information and Software Technology, 2012

Peter Lubbers & Frank Greco

Lately there has been a lot of buzz around HTML5 Web Sockets, which defines a full-duplex communication channel that operates through a single socket over the Web. HTML5 Web Sockets is not just another incremental enhancement to conventional HTTP communications; it represents a colossal advance, especially for real-time, event-driven web applications.

HTML5 Web Sockets provides such a dramatic improvement from the old, convoluted "hacks" that are used to simulate a full-duplex connection in a browser that it prompted Google's Ian Hickson—the HTML5 specification lead—to say:

"Reducing kilobytes of data to 2 bytes…and reducing latency from 150ms to 50ms is far more than marginal. In fact, these two factors alone are enough to make Web Sockets seriously interesting to Google."

Let's take a look at how HTML5 Web Sockets can offer such an incredibly dramatic reduction of unnecessary network traffic and latency by comparing it to conventional solutions.

Polling, Long-Polling, and Streaming—Headache 2.0

Normally when a browser visits a web page, an HTTP request is sent to the web server that hosts that page. The web server acknowledges this request and sends back the response. In many cases—for example, for stock prices, news reports, ticket sales, traffic patterns, medical device readings, and so on—the response could be stale by the time the browser renders the page. If you want to get the most up-to-date "real-time" information, you can constantly refresh that page manually, but that's obviously not a great solution.

Current attempts to provide real-time web applications largely revolve around polling and other server-side push technologies, the most notable of which is Comet, which delays the completion of an HTTP response to deliver messages to the client. Comet-based push is generally implemented in JavaScript and uses connection strategies such as long-polling or streaming.

With polling, the browser sends HTTP requests at regular intervals and immediately receives a response. This technique was the first attempt for the browser to deliver real-time information. Obviously, this is a good solution if the exact interval of message delivery is known, because you can synchronize the client request to occur only when information is available on the server. However, real-time data is often not that predictable, making unnecessary requests inevitable and as a result, many connections are opened and closed needlessly in low-message-rate situations.

With long-polling, the browser sends a request to the server and the server keeps the request open for a set period. If a notification is received within that period, a response containing the message is sent to the client. If a notification is not received within the set time period, the server sends a response to terminate the open request. It is important to understand, however, that when you have a high message volume, long-polling does not provide any substantial performance improvements over traditional polling. In fact, it could be worse, because the long-polling might spin out of control into an unthrottled, continuous loop of immediate polls.

With streaming, the browser sends a complete request, but the server sends and maintains an open response that is continuously updated and kept open indefinitely (or for a set period of time). The response is then updated whenever a message is ready to be sent, but the server never signals to complete the response, thus keeping the connection open to deliver future messages. However, since streaming is still encapsulated in HTTP, intervening firewalls and proxy servers may choose to buffer the response, increasing the latency of the message delivery. Therefore, many streaming Comet solutions fall back to long-polling in case a buffering proxy server is detected. Alternatively, TLS (SSL) connections can be used to shield the response from being buffered, but in that case the setup and tear down of each connection taxes the available server resources more heavily.

Ultimately, all of these methods for providing real-time data involve HTTP request and response headers, which contain lots of additional, unnecessary header data and introduce latency. On top of that, full-duplex connectivity requires more than just the downstream connection from server to client. In an effort to simulate full-duplex communication over half-duplex HTTP, many of today's solutions use two connections: one for the downstream and one for the upstream. The maintenance and coordination of these two connections introduces significant overhead in terms of resource consumption and adds lots of complexity. Simply put, HTTP wasn't designed for real-time, full-duplex communication as you can see in the following figure, which shows the complexities associated with building a Comet web application that displays real-time data from a back-end data source using a publish/subscribe model over half-duplex HTTP.

It gets even worse when you try to scale out those Comet solutions to the masses. Simulating bi-directional browser communication over HTTP is error-prone and complex and all that complexity does not scale. Even though your end users might be enjoying something that looks like a real-time web application, this "real-time" experience has an outrageously high price tag. It's a price that you will pay in additional latency, unnecessary network traffic and a drag on CPU performance.

HTML5 Web Sockets to the Rescue!

Defined in the Communications section of the HTML5 specification, HTML5 Web Sockets represents the next evolution of web communications—a full-duplex, bidirectional communications channel that operates through a single socket over the Web. HTML5 Web Sockets provides a true standard that you can use to build scalable, real-time web applications. In addition, since it provides a socket that is native to the browser, it eliminates many of the problems Comet solutions are prone to. Web Sockets removes the overhead and dramatically reduces complexity.

To establish a WebSocket connection, the client and server upgrade from the HTTP protocol to the WebSocket protocol during their initial handshake, as shown in the following example:

Example 1—The WebSocket handshake (browser request and server response)

```
GET /text HTTP/1.1\r\n Upgrade: WebSocket\r\n Connection: Upgrade\r\n Host:
www.websocket.org\r\n …\r\n
HTTP/1.1 101 WebSocket Protocol Handshake\r\n Upgrade: WebSocket\r\n
Connection: Upgrade\r\n …\r\n
```

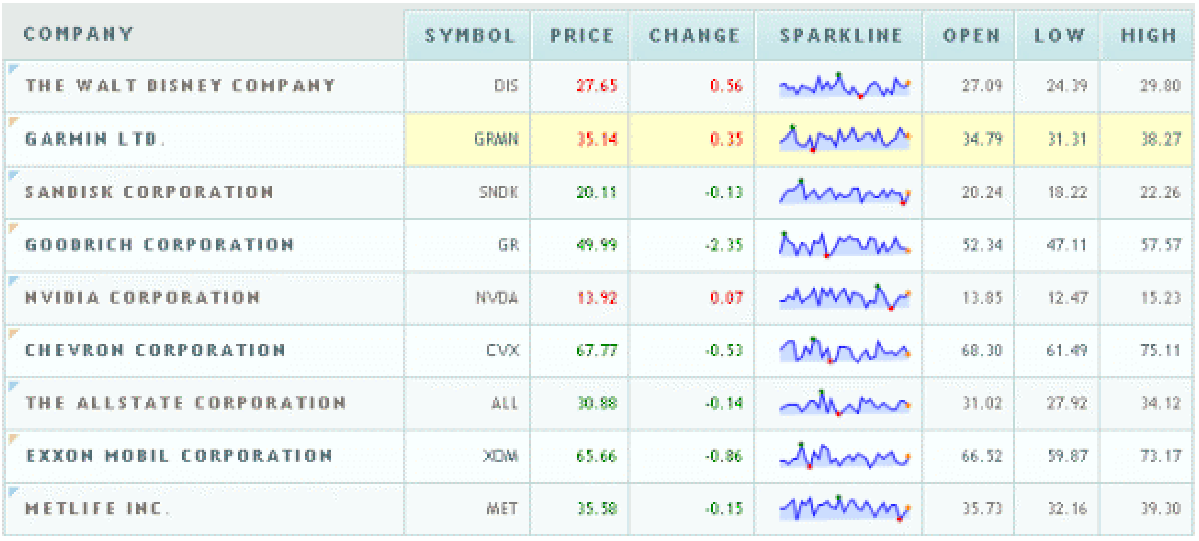
Once established, WebSocket data frames can be sent back and forth between the client and the server in full-duplex mode. Both text and binary frames can be sent full-duplex, in either direction at the same time. The data is minimally framed with just two bytes. In the case of text frames, each frame starts with a 0x00 byte, ends with a 0xFF byte, and contains UTF-8 data in between. WebSocket text frames use a terminator, while binary frames use a length prefix.

Note: although the Web Sockets protocol is ready to support a diverse set of clients, it cannot deliver raw binary data to JavaScript, because JavaScript does not support a byte type. Therefore, binary data is ignored if the client is JavaScript—but it can be delivered to other clients that support it.

The Showdown: Comet vs. HTML5 Web Sockets

So how dramatic is that reduction in unnecessary network traffic and latency? Let's compare a polling application and a WebSocket application side by side.

For the polling example, I created a simple web application in which a web page requests real-time stock data from a RabbitMQ message broker using a traditional publish/subscribe model. It does this by polling a Java Servlet that is hosted on a web server. The RabbitMQ message broker receives data from a fictitious stock price feed with continuously updating prices. The web page connects and subscribes to a specific stock channel (a topic on the message broker) and uses an XMLHttpRequest to poll for updates once per second. When updates are received, some calculations are performed and the stock data is shown in a table as shown in the following image.



COMPANY	SYMBOL	PRICE	CHANGE	SPARKLINE	OPEN	LOW	HIGH
THE WALT DISNEY COMPANY	DIS	27.65	0.56		27.09	24.39	29.80
GARMIN LTD.	GRMN	35.14	0.35		34.79	31.31	38.27
SANDISK CORPORATION	SNDK	20.11	-0.13		20.24	18.22	22.26
GOODRICH CORPORATION	GR	49.99	-2.35		52.34	47.11	57.57
NVIDIA CORPORATION	NVDA	13.92	0.07		13.85	12.47	15.23
CHEVRON CORPORATION	CVX	67.77	-0.53		68.30	61.49	75.11
THE ALLSTATE CORPORATION	ALL	30.88	-0.14		31.02	27.92	34.12
EXXON MOBIL CORPORATION	XOM	65.66	-0.86		66.52	59.87	73.17
METLIFE INC.	MET	35.58	-0.15		35.73	32.16	39.30

Figure 1—A JavaScript stock ticker application

Note: The back-end stock feed actually produces a lot of stock price updates per second, so using polling at one-second intervals is actually more prudent than using a Comet long-polling solution, which would result in a series of continuous polls. Polling effectively throttles the incoming updates here.

It all looks great, but a look under the hood reveals there are some serious issues with this application. For example, in Mozilla Firefox with Firebug (a Firefox add-on that allows you to debug web pages and monitor the time it takes to load pages and execute scripts), you can see that GET requests hammer the server at one-second intervals. Turning on Live HTTP Headers(another Firefox add-on that shows live HTTP header traffic) reveals the shocking amount of header overhead that is associated with each request. The following two examples show the HTTP header data for just a single request and response.

Example 2—HTTP request header

```
GET /PollingStock//PollingStock HTTP/1.1
 Host: localhost:8080
 User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US; rv:1.9.1.5)
Gecko/20091102 Firefox/3.5.5
 Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
 Accept-Language: en-us
 Accept-Encoding: gzip,deflate
 Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
 Keep-Alive: 300
 Connection: keep-alive
 Referer: http://www.example.com/PollingStock/
 Cookie: showInheritedConstant=false; showInheritedProtectedConstant=false;
showInheritedProperty=false; showInheritedProtectedProperty=false;
showInheritedMethod=false; showInheritedProtectedMethod=false;
showInheritedEvent=false; showInheritedStyle=false; showInheritedEffect=false
```

Example 3—HTTP response header

```
HTTP/1.x 200 OK
 X-Powered-By: Servlet/2.5
 Server: Sun Java System Application Server 9.1_02
 Content-Type: text/html;charset=UTF-8
 Content-Length: 21
 Date: Sat, 07 Nov 2009 00:32:46 GMT
```

Just for fun, I counted all the characters. The total HTTP request and response header information overhead contains 871 bytes and that does not even include any data! Of course, this is just an example and you can have less than 871 bytes of header data, but I have also seen cases where the header data exceeded 2000 bytes. In this example application, the data for a typical stock topic message is only about 20 characters long. As you can see, it is effectively drowned out by the excessive header information, which was not even required in the first place!

So, what happens when you deploy this application to a large number of users? Let's take a look at the network throughput for just the HTTP request and response header data associated with this polling application in three different use cases.

Use case A: 1,000 clients polling every second: Network throughput is (871 x 1,000) = 871,000 bytes = 6,968,000 bits per second (6.6 Mbps)

Use case B: 10,000 clients polling every second: Network throughput is (871 x 10,000) = 8,710,000 bytes = 69,680,000 bits per second (66 Mbps)

Use case C: 100,000 clients polling every 1 second: Network throughput is (871 x 100,000) = 87,100,000 bytes = 696,800,000 bits per second (665 Mbps)

That's an enormous amount of unnecessary network throughput! If only we could just get the essential data over the wire. Well, guess what? You can with HTML5 Web Sockets! I rebuilt the application to use HTML5 Web Sockets, adding an event handler to the web page to asynchronously listen for stock update messages from the message broker (check out the many how-tos and tutorials on tech.kaazing.com/documentation/ for more information on how to build a WebSocket application). Each of these messages is a WebSocket frame that has just two bytes of overhead (instead of 871)! Take a look at how that affects the network throughput overhead in our three use cases.

Use case A: 1,000 clients receive 1 message per second: Network throughput is (2 x 1,000) = 2,000 bytes = 16,000 bits per second (0.015 Mbps)

Use case B: 10,000 clients receive 1 message per second: Network throughput is (2 x 10,000) = 20,000 bytes = 160,000 bits per second (0.153 Mbps)

Use case C: 100,000 clients receive 1 message per second: Network throughput is (2 x 100,000) = 200,000 bytes = 1,600,000 bits per second (1.526 Mbps)

As you can see in the following figure, HTML5 Web Sockets provide a dramatic reduction of unnecessary network traffic compared to the polling solution.

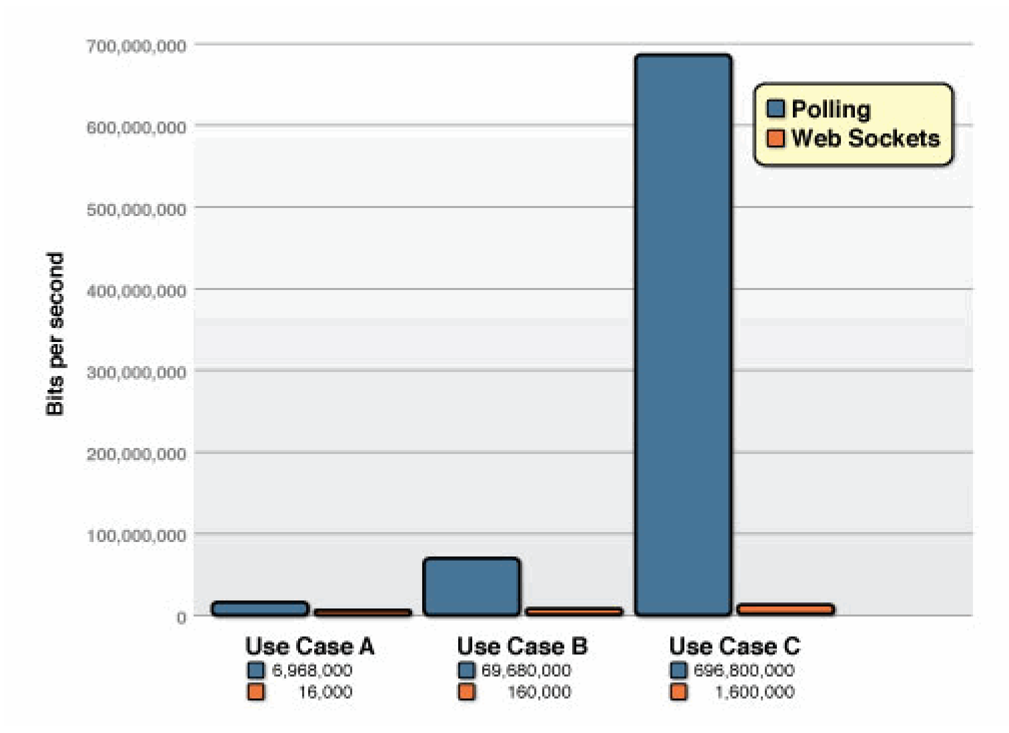


Figure 2—Comparison of the unnecessary network throughput overhead between the polling and the WebSocket applications

And what about the reduction in latency? Take a look at the following figure. In the top half, you can see the latency of the half-duplex polling solution. If we assume, for this example, that it takes 50 milliseconds for a message to travel from the server to the browser, then the polling application introduces a lot of extra latency, because a new request has to be sent to the server when the response is complete. This new request takes another 50ms and during this time the server cannot send any messages to the browser, resulting in additional server memory consumption.

In the bottom half of the figure, you see the reduction in latency provided by the WebSocket solution. Once the connection is upgraded to WebSocket, messages can flow from the server to the browser the moment they arrive. It still takes 50 ms for messages to travel from the server to the browser, but the WebSocket connection remains open so there is no need to send another request to the server.

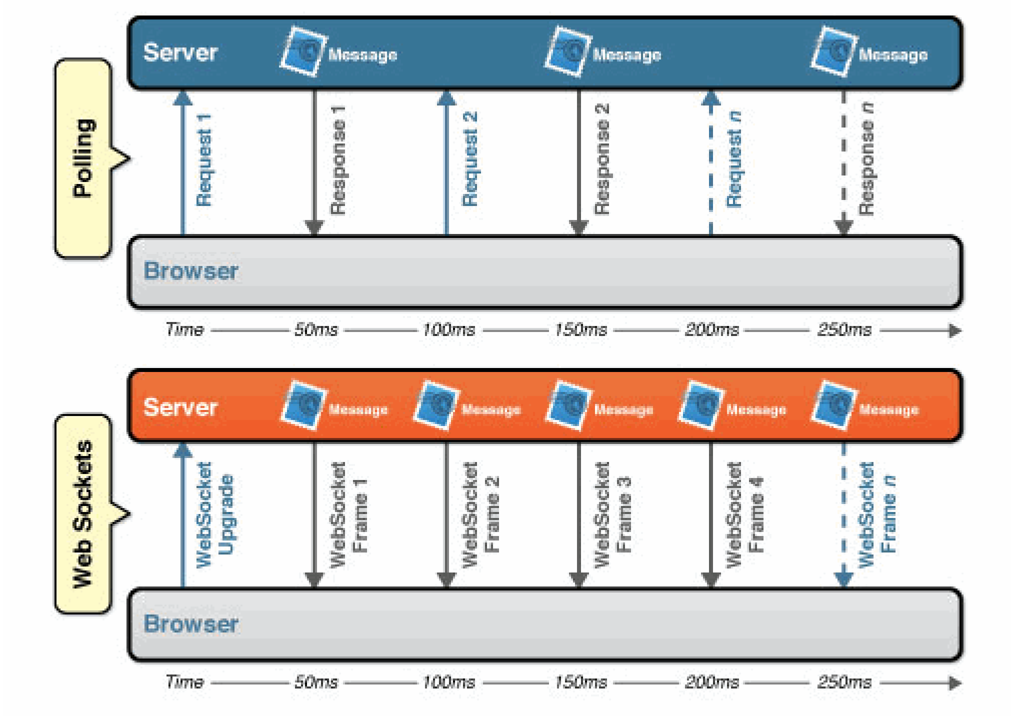


Figure 3—Latency comparison between the polling and WebSocket applications

HTML5 Web Sockets and the Kaazing WebSocket Gateway

Today, only Google's Chrome browser supports HTML5 Web Sockets natively, but other browsers will soon follow. To work around that limitation, however, Kaazing WebSocket Gateway provides complete WebSocket emulation for all the older browsers (I.E. 5.5+, Firefox 1.5+, Safari 3.0+, and Opera 9.5+), so you can start using the HTML5 WebSocket APIs today.

WebSocket is great, but what you can do once you have a full-duplex socket connection available in your browser is even greater. To leverage the full power of HTML5 Web Sockets, Kaazing provides a ByteSocket library for binary communication and higher-level libraries for protocols like Stomp, AMQP, XMPP, IRC and more, built on top of WebSocket.

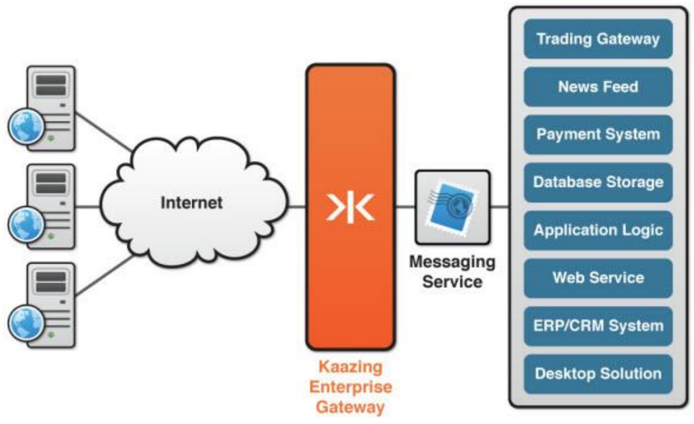


Figure 4—Kaazing WebSocket Gateway extends TCP-based messaging to the browser with ultra high performance

Summary

HTML5 Web Sockets provides an enormous step forward in the scalability of the real-time web. As you have seen in this article, HTML5 Web Sockets can provide a 500:1 or—depending on the size of the HTTP headers—even a 1000:1 reduction in unnecessary HTTP header traffic and 3:1 reduction in latency. That is not just an incremental improvement; that is a revolutionary jump—a quantum leap!

Kaazing WebSocket Gateway makes HTML5 WebSocket code work in all the browsers today, while providing additional protocol libraries that allow you to harness the full power of the full-duplex socket connection that HTML5 Web Sockets provides and communicate directly to back-end services.