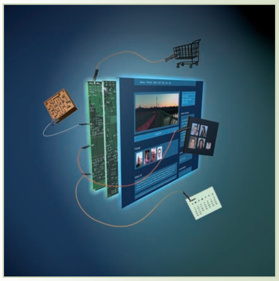

Communicating and Displaying Real-Time Data with WebSocket

Internet communication provides a convenient, hyperlinked, stateless exchange of information, but can be problematic when real-time data exchange is needed. The WebSocket protocol reduces Internet communication overhead and provides efficient, stateful communication between Web servers and clients. To determine whether WebSocket communication is faster than HTTP polling, the authors built a Web application to measure the one-way transmission latency of sending real-time wind sensor data at a rate of 4 Hz. They implemented a Jetty servlet to upgrade an HTTP connection to a WebSocket connection. Here, they compare the WebSocket protocol latency to HTTP polling and long polling.

Victoria Pimentel
Universidad Simón Bolívar

Bradford G. Nickerson
University of New Brunswick

Latency is a significant issue in applications such as networked control systems, where update frequencies of 10 to 500 milliseconds (ms) are required for adequate control of industrial processes.[1] Closed-loop control over the Internet is possible[2] by modeling the roundtrip delay and using UDP to consider only the most recent data, possibly discarding delayed packets. When an application must provide real-time data over an Internet connection in a peer-to-peer fashion, however (as when delivering real-time stock quotes or medical signals remotely for further processing), then latency becomes very important.

HTTP polling is considered a good solution for delivering real-time information if the message delivery interval is known — that is, when the data transmission rate is constant, as when transmitting sensor readings such as hourly temperature or water level. In such cases, the application developer can synchronize the client to request data when it's known to be available. When the rate increases, however, the overhead inherent to HTTP polling repeats significant header information, thus increasing latency. Earlier research posits that HTTP wasn't designed for real-time, full-duplex communication due to the complexity of real-time HTTP Web applications.[3] Thus, HTTP can simulate real-time communication only with a high price — increased latency and high network traffic.

Related Work in WebSocket Usage

Many researchers have tested and continue to test WebSocket usage for real-time applications. Bijin Chen and Zhiqi Xu have developed a framework that uses the WebSocket protocol for browser-based multiplayer online games.[1] They used a WebSocket implementation and evaluated performance in a LAN Ethernet network using Wireshark software to capture and analyze the size of IP packets traveling on the network. With a time interval of 50 milliseconds between updates of three game clients' states, their testing showed that the WebSocket protocol was sufficient to handle a server load of 96,257 bytes (758 packets) per second.

Peter Lubbers and Frank Greco compare the WebSocket protocol with HTTP polling in an application that updates stock quotes every second.[2] Their analysis shows a three-to-one reduction in latency and up to a 500-to-one reduction in HTTP header traffic. One question this research hasn't answered, however, is whether the advantage of less overhead for WebSocket protocol communication persists over a wide area network.

Our investigation in the main text explores the WebSocket protocol's efficiency over long distances via the Internet. We performed experimental validation with clients located in different countries and at different times of day to probe a variety of network conditions.

References

1. B. Chen and Z. Xu, "A Framework for Browser-Based Multiplayer Online Games Using Webgl and Websocket," *Proc. Int'l Conf. Multimedia Technology* (ICMT 11), IEEE Press, 2011, pp. 471–474.
2. P. Lubbers and F. Greco, "HTML5 Web Sockets: A Quantum Leap in Scalability for the Web," *SOA World Magazine*, Mar. 2010; http://soa.sys-con.com/node/1315473.

Long polling is a variation on HTTP polling that emulates the information push from a server to a client. The Comet Web application model,[4] for instance, was designed to push data from a server to a browser without a browser HTTP request, but is generally implemented using long polling to accommodate multiple browsers. Long polling isn't believed to provide any substantial improvement over traditional polling.[5]

The WebSocket protocol enables full-duplex communication between a client and a remote host over a single TCP socket.[6] The WebSocket API is currently a W3C working draft,[7] but the protocol is estimated to provide a three-to-one reduction in latency against half-duplex HTTP polling applications.[5]

Here, we compare the one-way transmission latency of WebSocket, long polling, and the best-case scenario for HTTP polling in a real-time application (see the "Related Work in WebSocket Usage" sidebar for other research in this area). We experimentally validate latency behavior at a 4-Hz rate for the low-volume communication (roughly 100 bytes per second of sensor data) typical of real-time sensor networks.

Web Client-Server Communication

To evaluate the Internet's effectiveness for real-time data exchange, we compare WebSocket communication with HTTP. We didn't consider other Internet protocols, such as UDP,[8] because they're designed for streaming real-time data when the newest data is more important and allowing older information to be dropped.

HTTP Polling

HTTP polling consists of a sequence of request-response messages. The client sends a request to a server. Upon receiving this request, the server responds with a new message, if there is one, or with an empty response if no new message is available for that client. After a short time Δ, called the *polling interval*, the client polls the server again to see if any new messages are available. Various applications including chat, online games, and text messaging use HTTP polling.

HTTP Long Polling

One weakness associated with polling is the number of unnecessary requests made to the server when it has no new messages for a client. Long polling emerged as a variation on the polling technique that efficiently handles the information push from servers to clients. With long polling, the server doesn't send an empty response immediately after realizing that no new messages are available for a client. Instead, the server holds the request until a new message is available or a timeout expires. This reduces the number of client requests when no new messages are available.

WebSocket

With continuous polling, an application must repeat HTTP headers in each request from the client and each response from the server. Depending on the application, this can lead to increased communication overhead. The WebSocket protocol provides a full-duplex, bidirectional communication channel that operates through a single socket over the Web and can help build scalable, real-time Web applications.[5]

The WebSocket protocol has two parts. The *handshake* consists of a message from the client and the handshake response from the server. The second part is *data transfer*. Jetty's implementation of the WebSocket API is fully integrated into the Jetty HTTP server and servlet containers (see http://jetty.codehaus.org/jetty). Thus, a Jetty servlet can process and accept a request to upgrade an HTTP connection to a WebSocket connection. Further details on the WebSocket communication process are available in our prior work.[9]

Architecture

Our WindComm Web application using the WebSocket protocol has three main components: the wind sensor, the base station computer (server), and the client. The base station computer employs a Jetty server running a Web application called WindComm. This application communicates with the sensor and manages HTTP and WebSocket requests from clients. A client accesses the Web application to see real-time wind sensor data using a Web browser that supports the WebSocket protocol and HTML5's Canvas element.

Wind Sensor

The Gill WindSonic is a robust, ultrasonic wind sensor with no moving parts that measures wind direction and speed (see www.gill.co.uk/products/anemometer/windsonic.html). We connected the WindSonic to a base station computer through an RS232 output cable connected to a USB serial port in the base station computer via an adapter. We simulated dynamic wind with an oscillating fan.

WindSonic operates in three modes: continuous, polled, and configuration. We used continuous mode and a data rate of 4 Hz to send 22-byte messages continuously.

Base Station Computer

The base station computer runs the WindComm Web application implementing a Jetty servlet. The application communicates with the sensor using the RXTX Java library (http://rxtx.qbang.org/wiki/index.php/Main_Page) to access the computer serial port. WindComm provides a near real-time channel for sensor data and must keep up with the sensor's 4-Hz output rate. We implemented the WindComm Web application in three versions. The first, called WindComm, uses Jetty's implementation of the HTML WebSocket protocol. The second, LongPollingWindComm, implements HTTP long polling, and the third, PollingWindComm, uses HTTP polling. In all three approaches, we implemented a thread to establish and maintain communication with the wind sensor through the base station computer serial port.

For LongPollingWindComm, we used Jetty's Continuations interface, which lets the servlet suspend and hold a client request until an event occurs or a timeout expires. For LongPollingWindComm, the event is a new sensor measurement, and we set the timeout to 300 ms, which is 50 ms more than the sensor's output rate.

In PollingWindComm, the servlet doesn't hold the client request. Setting the timeout to 250 ms would assume that the latency is 0 ms. We know the latency is significantly higher than this, so setting Δ to 250 ms would result in PollingWindComm running very slowly because it would take longer to process the accumulating queue of sensor observations. Thus, we set the polling interval Δ of the client to 150 ms, 100 ms less than the sensor's output rate. We also considered the time that the client takes to parse and display a sensor observation received from the server before polling the server again. We don't count this parse-and-display time in the latency observations, but we must account for it when setting the polling interval.

Experimental Design

Our experiments compare one-way latency between a client and our server for the WindComm, LongPollingWindComm, and PollingWindComm Web applications. Figure 1 shows a timeline with marked events that are relevant to our tests. For LongPollingWindComm, the timeline is similar to the polling timeline, except that t_2 doesn't necessarily occur after t_1 or t_0. If a client request has been held, after t_1 the servlet resumes using the Continuations interface, and sends the packet to the client immediately. The servlet keeps measured data that it hasn't yet transmitted in a buffer. It sends all buffered

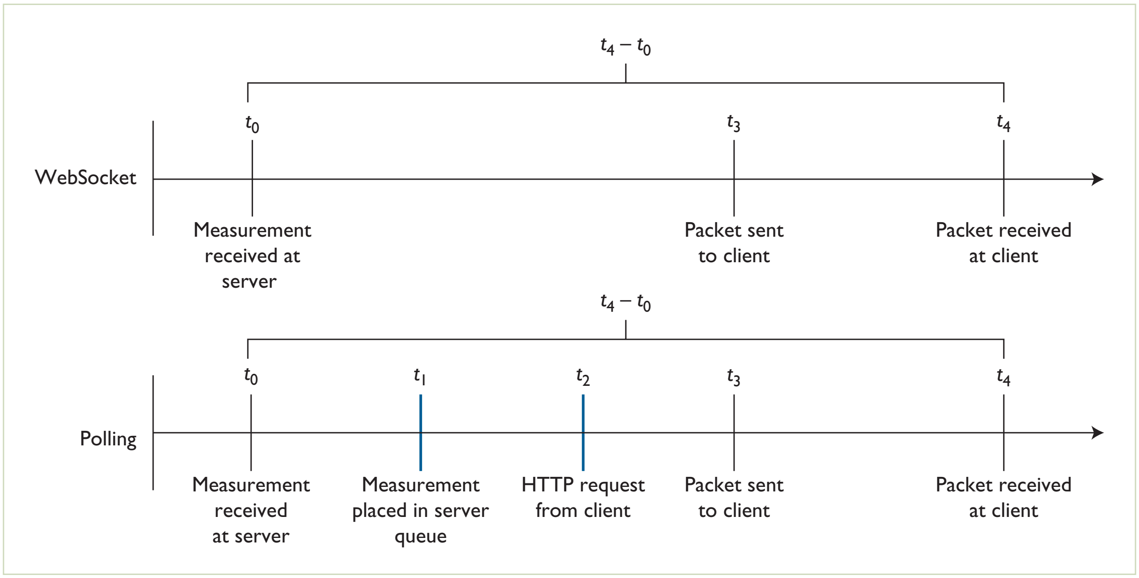
$t_4 - t_0$
t_0
t_3
t_4
WebSocket
Measurement received at server
Packet sent to client
Packet received at client
$t_4 - t_0$
t_0
t_1
t_2
t_3
t_4
Polling
Measurement received at server
Measurement placed in server queue
HTTP request from client
Packet sent to client
Packet received at client

Figure 1. The time epochs at which we recorded time stamps to evaluate latency. In all cases, latency is defined as $t_4 - t_0$*, and doesn't include the time to parse and display a sensor measurement.*

data each time a poll occurs for either polling version.

Our definition of latency for all three versions of the WindComm Web application is $t_4 - t_0$. To report this one-way latency, the application takes a time stamp at the server for t_0 and a second one at the client for t_4. To make the time stamps comparable, the client and server must be synchronized.

Time Synchronization

The Network Time Protocol (NTP) is widely used to synchronize computer clocks over the Internet.[10] The NTP packet is a UDP datagram carried on port 123. For Linux, NTP is implemented as a daemon to run continuously. This daemon, NTPd, maintains the system time synchronized with NTP time servers. We configured NTPd on the base station computer and all four client test computers to synchronize with an NTP time server. Immediately before starting a test, we (or a colleague at the client location) ran the command `"ntpq -p"` in the client and the server until they each reported an offset magnitude below 2 ms. The server always reported an offset below 1 ms. We repeated the command after each test as well to make sure the offset remained below 2 ms. After synchronizing the time in this fashion, the client directed its HTML5-capable browser (Firefox 6.0.2 or later) to one of the three Web applications by entering the appropriate URL (such as http://131.202.243.62:8080/WindComm/).

As soon as the client receives a message, it takes a local time stamp. The client then parses the message received, extracts the server time stamp, calculates the latency, and saves it in an array. When the array of 1,200 latencies is filled, the test ends, and the client sends the array's contents to the server. We chose an array size of 1,200 to correspond to approximately five minutes of measurements at a continuous 4-Hz rate.

Testing

Our tests ran WindComm, LongPollingWindComm, and PollingWindComm one after another at three different local times until each application successfully delivered 1,200 messages. The total time taken to run three applications for each test was approximately 15 minutes, plus the latency, the time to start applications, and the time to report the results from the client to the base station. We planned the first test for around 8:00 a.m. (not busy), the second test for around 1:00 p.m. (normal traffic), and the third test around 8:00 p.m. (busy). We chose these times to vary the network state. Although it would have been interesting to run the test interspersing messages — that is, one message from WindComm followed by one from LongPollingWindComm followed by one from PollingWindComm to provide a more comparable network state for each protocol — this wasn't possible. Only one running process (one Web application) in our base station computer can access the wind sensor at a time.

We ran the tests between our server located at the University of New Brunswick in eastern Canada, with clients in Edmonton, Canada; Caracas, Venezuela; Lund, Sweden; and Nagaoka, Japan. Note that, except for Lund, all the clients were located on a university campus. This means that our test data was likely routed over the research networks connecting university campuses and not over the commercial Internet. The client in Lund was located in a company office building.

Results

Table 1 shows the results of our evaluation. We ran a total of 12 tests for each method — WebSocket (WS), long polling (LP), and polling (P), repeating each test three times with the client in four countries. In all 36 test cases, the server delivered the 1,200 measurements to the client within 5 minutes and 1 second after starting the test. Table 1 reports the test start time, observed average latency μ (ms, for $N = 1{,}200$), the sample standard deviation s (ms), and the ratio r of $\frac{\mu_{LP}}{\mu_{WS}}$ or $\frac{\mu_P}{\mu_{WS}}$ for each of the tests. Tests in bold are those we selected for further analysis.

For the real-time, low-volume continuous data used here, all the tests showed that HTTP polling average latency is significantly higher (between 2.3 and 4.5 times higher) than either WebSocket or long polling. The WebSocket protocol can have a lower or higher average latency than long polling. Over longer distances (such as to Japan), the WebSocket protocol has significantly (between 3.8 and 4.0 times) lower average latency than long polling.

In the selected (bold) test results for Edmonton, we observe that polling has a 3.75 times longer average latency than the WebSocket protocol (151.3 versus 40.3 ms). A *difference of means* statistical test (with unknown and different population variances) indicates that the null hypothesis $H_0 : \mu_{WS} - \mu_P = 0$ is rejected at the 99 percent confidence level in favor of the alternate hypothesis $H_1 : \mu_{WS} - \mu_P < 0$. Thus, we have enough evidence to affirm that the WebSocket protocol is significantly faster than HTTP polling within Canada. In fact, all our statistical testing provides strong evidence that the WebSocket protocol always has significantly lower latency than polling for the low-volume, real-time data communication testing done here.

Long polling average latency for the 5-minute test period starting at 9:10 a.m. was only 1.0 ms longer than the WebSocket latency. Despite this, the null hypothesis $H_0 : \mu_{WS} - \mu_{LP} = 0$ is also rejected at the 99 percent confidence level in favor of the alternate hypothesis $H_1 : \mu_{WS} - \mu_{LP} < 0$. The difference in average latency of 1.0 ms is less than the time synchronization offset threshold of 2 ms. In all the Edmonton cases, long polling and WebSocket average latencies can be considered the same within experimental uncertainty.

The results for Caracas are essentially the same, except for the selected tests starting at 12:00 noon and 12:05 p.m. In this case, the null hypothesis $H_0 : \mu_{WS} - \mu_{LP} = 0$ can't be rejected at the 99 or 95 percent confidence levels in favor of the alternate hypothesis $H_1 : \mu_{WS} - \mu_{LP} \neq 0$. Our evidence indicates that, in this case, the WebSocket and long polling mean latencies are the same.

The selected results for Lund show the same trend as for Caracas — that is, the long polling average latency of 87.5 ms starting at 10:53 a.m. is 4.4 ms faster than the WebSocket average latency of 91.9 ms. In this case, the null hypothesis $H_0 : \mu_{WS} - \mu_{LP} = 0$ is rejected at the 99 percent confidence level in favor of $H_1 : \mu_{WS} - \mu_{LP} > 0$. Thus, we have enough evidence to affirm that the WebSocket average latency μ_{WS} is greater than the long polling average latency μ_{LP}.

All three test cases for Nagaoka are consistent. The long polling average latency is significantly (3.6 to 4.2 times) higher than the WebSocket average latency. Statistical testing shows that the null hypothesis $H_0 : \mu_{WS} - \mu_{LP} = 0$ is rejected at the 99 percent confidence level in favor of $H_1 : \mu_{WS} - \mu_{LP} < 0$ in all three cases. In one case (start times 11:22 and 11:28 a.m.), the long polling average latency of 647.0 ms exceeds that of the 584.3 ms polling average latency. The null hypothesis $H_0 : \mu_{LP} - \mu_P = 0$ is rejected at the 99 percent confidence level in favor of the alternate hypothesis $H_1 : \mu_{LP} - \mu_P > 0$.

Long Polling

To explain why long polling performs nearly as well as the WebSocket protocol in all but the Nagaoka test, we divided our results into three cases. The first case considers tests in which $\mu_{LP} \leq 125$ ms, the second tests where 125 ms $< \mu_{LP} \leq 250$ ms, and the third tests where $\mu_{LP} > 250$ ms.

Table 1. Evaluation results.[+]

Start	Method	Average latency (μ)	Standard deviation (s)	Ratio (r)
Edmonton, Canada				
9:04 a.m.	**WS**[*]	**40.3**	**7.6**	**1**
9:10 a.m.	**LP**	**41.3**	**0.86**	**1.02**
9:21 a.m.	**P**	**151.3**	**63.7**	**3.75**
1:04 p.m.	WS	39.7	2.8	1
1:10 p.m.	LP	40.7	2.4	1.02
1:16 p.m.	P	149.9	63.6	3.77
7:01 p.m.	WS	40.5	0.52	1
7:07 p.m.	LP	41.5	0.97	1.02
7:13 p.m.	P	150.4	63.4	3.72
Caracas, Venezuela				
10:25 a.m.	WS	122.9	48.2	1
10:32 a.m.	LP	131.5	66.3	1.07
10:45 a.m.	P	283.5	111.7	2.31
12:00 noon	**WS**	**124.0**	**51.02**	**1**
12:05 p.m.	**LP**	**121.4**	**50.8**	**0.98**
12:11 p.m.	P	298.0	114.1	2.40
7:00 p.m.	WS	98.8	1.37	1
7:08 p.m.	LP	106.1	41.2	1.07
7:16 p.m.	P	266.9	100.1	2.70
Lund, Sweden				
10:45 a.m.	**WS**	**91.92**	**2.74**	**1**
10:53 a.m.	**LP**	**87.5**	**2.26**	**0.95**
11:00 a.m.	P	241.4	88.4	2.63
3:43 p.m.	WS	95.3	2.08	1
3:51 p.m.	LP	97.3	1.72	1.02
3:58 p.m.	P	253.0	90.1	2.65
1:11 a.m.	WS	75.1	0.42	1
1:17 a.m.	LP	77.6	0.88	1.03
1:23 a.m.	P	225.1	84.8	3.00
Nagaoka, Japan				
11:16 a.m.	WS	153.4	133.3	1
11:22 a.m.	**LP**	**647.0**	**683.4**	**4.22**
11:28 a.m.	**P**	**584.3**	**508.8**	**3.81**
12:32 p.m.	WS	163.3	161.9	1
12:45 p.m.	LP	585.7	622.0	3.59
12:55 p.m.	P	699.5	682.8	4.28
9:28 p.m.	WS	156.2	176.0	1
9:33 p.m.	LP	623.3	654.6	3.99
9:40 p.m.	P	700.1	711.0	4.48

[+]$N = 1{,}200$ in all 36 tests; bold rows indicate tests selected for further analysis
[*]WS: WebSocket; LP: long polling; P: polling

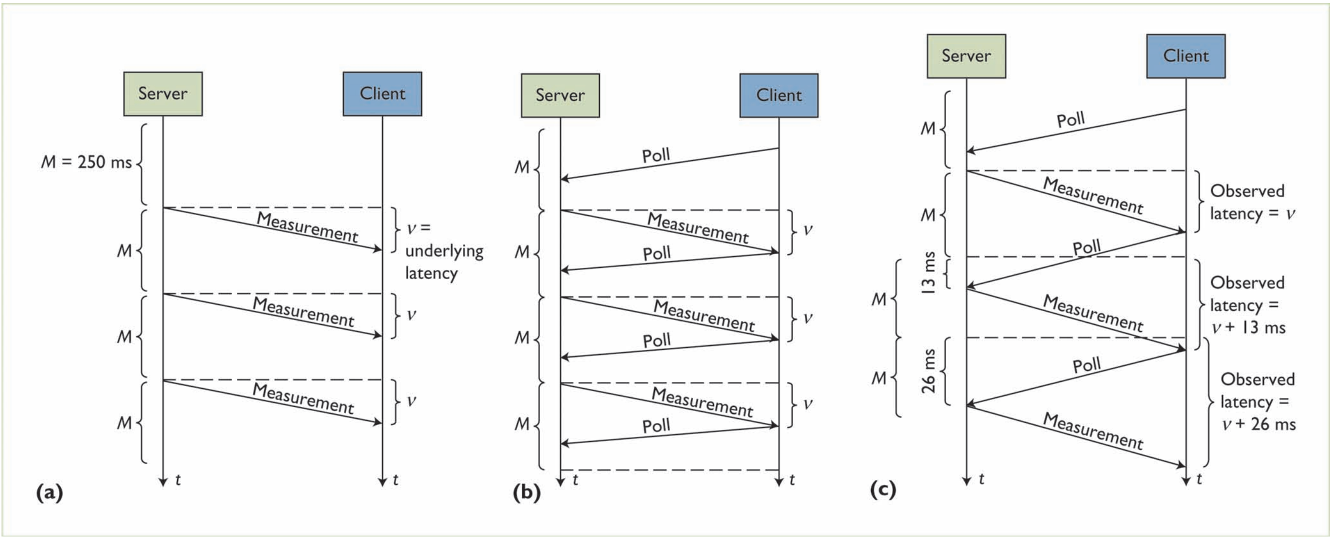

Figure 2. WindComm Web application communication behavior. We show (a) WebSocket behavior; (b) long polling behavior when $\mu_{LP} \leq 125$ ms; and (c) long polling behavior when $\mu_{LP} > 125$ ms. Measurements occur at a constant rate of one every M ms.

Figure 2a illustrates the WebSocket protocol behavior for the WindComm Web application once it has successfully established the socket connection. This behavior represents the one-way communication at a known constant rate (250 ms) between the server and a client. Each time the server receives a measurement from the sensor, the server immediately sends the measurement to the client.

Figure 2b shows the LongPollingWindComm behavior for the first case. This means that a measurement can travel to the client in fewer than 125 ms, and the next poll request can travel back to the server in fewer than 125 ms. As a consequence, a new poll request will be waiting at the server before the next sensor measurement arrives. In terms of latency, this behavior performs like the WebSocket protocol. The server can send a sensor reading to the client as soon as it receives the measurement and no accumulation of sensor readings exists in the server's queue. We observed this behavior in eight tests: all of the Lund and Edmonton tests and in the Caracas tests starting at 12:00 noon and 7:00 p.m. This corresponds to two-thirds of the tests in which long polling performed like the WebSocket protocol for one-way communication at a constant known rate.

Figure 2c shows WindComm Web application behavior for the second case. The client first polls the server. The client's request arrives at the server before or at the exact time a measurement is available. The server receives a sensor measurement, resumes the client's request, and sends the sensor message immediately. Latency is greater than 125 ms, which corresponds to half the 250-ms observation rate. The total of (response message travel time to client + poll request travel time to server) exceeds 250 ms. Thus, when the server receives a new poll request from the client, at least one sensor reading will be in the client's queue. For example, for the Caracas test starting at 10:32 a.m., the long polling mean latency reported is 131.5 ms. Figure 2c shows that by the time the server receives a second poll request, a sensor reading will exist in the server that's 13 ms old. This is because it takes 131.5 ms for the first measurement to travel to the client and another 131.5 ms for the second poll request to travel from the client to the server. On the server side, $263 - 250 = 13$ ms have passed since the server received a new sensor reading. This 13 ms will increase as the application runs. The server will receive the next request 26 ms after the latest measurement, and so on. When the client's poll request is delayed by 250 ms, a second sensor reading will be waiting in the client's queue; this reading will be sent along with the older measurement. The ratio r for long polling in this case (Venezuela, 10:32 a.m.) is only 1.07, and the difference of means is 8.6 ms.

A more detailed analysis revealed that the third quartile of the data is 125 ms, meaning that 75 percent of the data is below 125 ms. Thus, the second case applies to only 25 percent of the data from the Venezuela test at 10:32 a.m., and the long polling performance isn't significantly affected.

We observed the third case occurring in all three Nagaoka tests. Multiple measurements were queued at the server waiting for the poll request to arrive. In the worst case, μ_{LP} is $2v$, where v is the underlying latency, and the expected value of μ_{LP} is $\frac{3v}{2}$. The fact that Table 1's results for Nagaoka show values of μ_{LP} significantly longer than $2\mu_{WS}$ is likely due to the relatively high observed variance of μ_{LP}.

Polling

Only one result in all 12 tests showed polling performing better than long polling. In no test did polling perform better than the WebSocket protocol.

An average latency below 125 ms causes long polling to perform like the WebSocket protocol. This isn't the case for polling. Assume the underlying latency is v ms (as opposed to the observed average latency μ), and that the first request arrives at the server before a measurement is available. In polling, when no message is available for the client, the server sends an empty response at time t. The client receives the empty response at time $t + v$ ms (at the earliest). The server receives a new measurement at time $t + M$ ms (at the latest), where M is the time between measurements. The client waits $\Delta = 150$ ms (the polling interval) before polling the server again at time $t + v + \Delta$ ms. This request arrives at the server at time $t + 2v + \Delta$ ms, $2v + \Delta - M$ ms after the sensor takes the measurement.

As long as $2v + \Delta > M$, at least one message will be waiting in the server queue for the next request. If $2v + \Delta < M$, polling should be able to keep up with the real-time measurements. In the worst case, the server receives the poll request immediately before the sensor takes a measurement, and the polling observed latency is $3v + \Delta$. In the best case, a poll request arrives immediately after the sensor takes a measurement, and the observed latency is v. Assuming that the poll requests arrive in a uniform random fashion, this gives an expected value of $2v + \frac{\Delta}{2}$ for the observed polling latency. We see this expected behavior in the Edmonton results, where, assuming $v = 40$ ms, we have $2v + \frac{\Delta}{2} =$ 155 ms.

As we've demonstrated through testing, the WebSocket protocol can keep up with a continuous 4-Hz data rate in all four test locations we tried, giving an average latency of 40.3 ms within Canada, up to an average latency of 163.3 ms between eastern Canada (Fredericton, New Brunswick) and Japan. Regular HTTP polling can keep up with a 4-Hz continuous data rate within Canada (with an average latency of 150.5 ms) and between Canada and Sweden (with an average latency of 239.8 ms). As long as $2v + \Delta < M$, where v is the underlying latency, Δ is the polling interval, and M is the time between real-time measurements, we expect polling to be able to deliver the same performance as WebSocket or long polling. When the network distance increases, increasing v, HTTP polling is at a disadvantage, as illustrated by the average observed latency of 282.8 ms between eastern Canada and Venezuela, and 661.3 ms between eastern Canada and Japan. Additionally, the test results show that long polling can perform as well as the WebSocket protocol as long as the underlying latency is normally less than half the data measurement rate — that is, when $v < M/2$.

Using difference of means hypothesis testing for all 12 tests, we calculated that there are nine tests in which $\mu_{WS} < \mu_{LP}$, one test in which $\mu_{LP} < \mu_{WS}$ (although the difference of 4.4 ms is slight), and two tests in which $\mu_{WS} = \mu_{LP}$. When the packets must travel a long distance, or via a congested network, resulting in a network underlying latency that exceeds half the data measurement rate, then the WebSocket protocol is clearly a better choice.

Our results are based on a small sample of the possible Internet communication paths, and for a very limited time, but are likely to be a good estimate of how WebSocket communication works for low-volume continuous real-time traffic occurring at a rate of 4 Hz. The stateful approach that the WebSocket protocol uses does provide better latency (on average) for real-time Internet communication.

Acknowledgments

We thank the University of New Brunswick Faculty of Computer Science for their support of this research. We also thank our colleagues in Edmonton, Winnipeg, Caracas, Lund, and Nagaoka for making our experimental results possible. Thanks are also due to several anonymous reviewers whose comments helped to significantly improve the article.

References

1. J. Åkerberg, M.M. Gidlund, and M. Björkman, "Future Research Challenges in Wireless Sensor and Actuator Networks Targeting Industrial Automation," *Proc. 9th IEEE Int'l Conf. Industrial Informatics* (INDIN 11), IEEE Press, 2011, pp. 410–415.
2. W. Hu, G.-P. Liu, and D. Rees, "Networked Predictive Control over the Internet Using Round-Trip Delay Measurement," *IEEE Trans. Instrumentation and Measurement*, vol. 57, no. 10, 2008, pp. 2231–2241.
3. P. Lubbers, B. Albers, and F. Salim, *Pro HTML5 Programming: Powerful APIs for Richer Internet Application Development*, Apress, 2010.
4. R. Gravelle, "Comet Programming: Using Ajax to Simulate Server Push," Webreference tutorial, Mar. 2009; www.webreference.com/programming/javascript/rg28/index.html.
5. P. Lubbers and F. Greco, "HTML5 Web Sockets: A Quantum Leap in Scalability for the Web," *SOA World Magazine*, Mar. 2010; http://soa.sys-con.com/node/1315473.
6. I. Fette and A. Melnikov, "The Websocket Protocol," IETF Internet draft, work in progress, Dec. 2011.
7. I. Hickson, "The Websocket API," W3C candidate recommendation, Dec. 2011; www.w3.org/TR/websockets.
8. A.S. Tanenbaum, *Computer Networks*, 4th ed., Prentice Hall, 2003.
9. V. Pimentel and B.G. Nickerson, *Web Display of Real-Time Wind Sensor Data*, tech. report TR11-214, Faculty of Computer Science, Univ. of New Brunswick, Dec. 2011.
10. D. Mills et al., *Network Time Protocol Version 4: Protocol and Algorithms Specification*, IETF RFC 5905, June 2010; http://tools.ietf.org/html/rfc5905.

Victoria Pimentel is a student in the computing engineering program at the Universidad Simón Bolívar, Caracas, Venezuela. Her interests are software engineering and sensor networks. Her research on connecting a real-time ultrasonic wind sensor to the Web led to the results presented in this article, and piqued her interest in Web protocols and their performance. Pimentel recently completed an internship in the Faculty of Computer Science at the University of New Brunswick. Contact her at v.pimentel.gue@gmail.com.

Bradford G. Nickerson is the Assistant Dean, Research and Outreach, for the University of New Brunswick's Faculty of Computer Science as well as the software engineering program coordinator. His research interests include spatial data structures and sensor Web systems. Nickerson has a PhD in computer and systems engineering from Rensselear Polytechnic Institute. He's a member of the IEEE Computer Society, the Canadian Information Processing Society, and the Association of Professional Engineers and Geoscientists of New Brunswick. Contact him at bgn@unb.ca.