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.

1089-7801/12/$31.00 © 2012 IEEE Published by the IEEE Computer Society

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

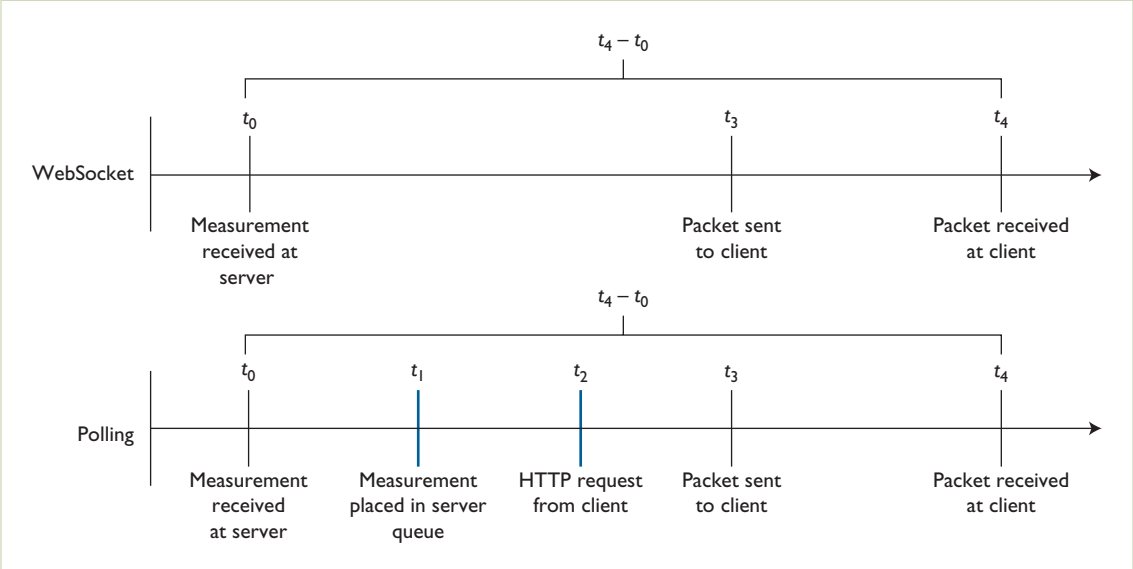

Figure 1. The time epochs at which we recorded time stamps to evaluate latency. In all cases, latency is defined as $t_4 - t_0$, and doesn't include the time to parse and display a sensor measurement.

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.

DOCKET ALARM

Explore Litigation Insights

Docket Alarm provides insights to develop a more informed litigation strategy and the peace of mind of knowing you're on top of things.

Real-Time Litigation Alerts

Keep your litigation team up-to-date with **real-time alerts** and advanced team management tools built for the enterprise, all while greatly reducing PACER spend.

Our comprehensive service means we can handle Federal, State, and Administrative courts across the country.

Advanced Docket Research

With over 230 million records, Docket Alarm's cloud-native docket research platform finds what other services can't. Coverage includes Federal, State, plus PTAB, TTAB, ITC and NLRB decisions, all in one place.

Identify arguments that have been successful in the past with full text, pinpoint searching. Link to case law cited within any court document via Fastcase.

Analytics At Your Fingertips

Learn what happened the last time a particular judge, opposing counsel or company faced cases similar to yours.

Advanced out-of-the-box PTAB and TTAB analytics are always at your fingertips.

API

Docket Alarm offers a powerful API (application programming interface) to developers that want to integrate case filings into their apps.

LAW FIRMS
Build custom dashboards for your attorneys and clients with live data direct from the court.

Automate many repetitive legal tasks like conflict checks, document management, and marketing.

FINANCIAL INSTITUTIONS
Litigation and bankruptcy checks for companies and debtors.

E-DISCOVERY AND LEGAL VENDORS
Sync your system to PACER to automate legal marketing.