Survey: WebSocket

Thibault Gérondal, Michaël Heraly

Abstract—The most common way to get information and to communicate through the internet is via the Hypertext Transfer Protocol (HTTP). Over time, the shared media sent through this communication system has evolved, from text to images and videos. Interactions between clients and servers have evolved as well. We went from passive customers who receive information to active clients wanting to communicate in real-time. The original HTTP was never designed to achieve those needs. The market found some tricks to bypass these limitations. However, the HTML5 initiative introduced a real solution to this problem : WebSocket. This solution brings sockets to the web, so that full-duplex communications can be established between clients and servers.

In this survey paper, we describe the older techniques that were used to achieve a full-duplex communication before describing the WebSocket protocol itself. Then, we explore some experiments that show how WebSocket is more efficient than these older methods. Finally, we discuss the security issues of WebSocket.

1 Introduction

The Hypertext Transfer Protocol (HTTP) is a stateless request-response protocol in the server-client computing model. The client submits an HTTP request message and the server provides a response (HTML files, images, etc.). With the growing popularity of the web, the number of web applications has risen significantly, and the need of interactivity between the client and the server has also increased. In the original HTTP specifications, interactivity was only possible by loading an entire page in order to upload information to the server or to receive new information from it. New features of web browsers have arisen in order to successfully add interactivity without (re)loading pages [1]. Among these, two Javascript API were implemented successively in web browsers : XMLHttpRequest and WebSocket.

2 XMLHttpRequest

One of the first and most used solutions to add real-time interactivity is the XMLHttpRequest (XHR) Object, which is a Javascript API that permits the browser to send an HTTP request to query a resource from a distant server. Despite the name of this API, the fetched resource does not have to be an XML file. In fact, JSON (JavaScript Object Notation) is generally used as it is easier to parse in Javascript [2]. This technique is great to send data to the server but not for receiving new data. If the client wants to keep the information up to date from the server, it will have to poll the server periodically. This behavior generates a lot of overhead as each request and response will have a full HTTP header and a lot of those requests might return no new data. This technique thus proves to be quite ineffective.

2.1 The long polling version

The XHR long polling exploits a loophole inside the HTTP/1.1 specifications. After receiving an HTTP request, a web server is not required to respond immediately, it can defer the reply for a few instants [2]. With this technique, the client does not have to poll the server periodically as the web server holds the HTTP request open until it has new data to send to the client, or until a timeout occurs. This reduces the amount of useless data transfers and improves the update delay.

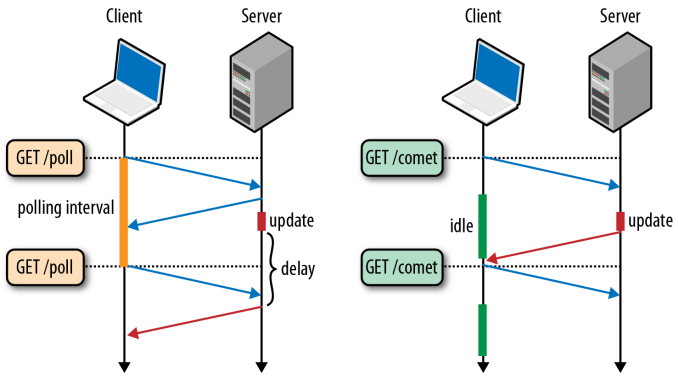

Figure 1. periodic polling (left) versus long polling (right) (from [3])

In Fig. 1, on the left, the client periodically polls the server to retrieve updates. This is shown on the figure as the "polling interval". When an update occurs on the server side, the client has to wait for the next polling to become aware of it. On the right, with long polling, the server keeps the connection open until the update occurs. The periodic polling introduces a delay that long polling can avoid. Also, the first poll of the periodic polling is superfluous as it generates data on the network without any real new input.

3 WebSocket protocol

The WebSocket protocol was standardized by the IETF as RFC 6455 in 2011 [4]. This protocol was designed to bring bidirectional, message-oriented streaming of texts and binary data between web browsers and web servers.

As it was designed for the Web, this protocol copes with existing HTTP infrastructures, which means working over HTTP ports 80 and 443.As the WebSocket protocol is

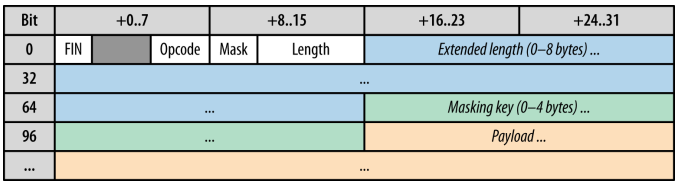

Figure 2. WebSocket frame structure (from [5])

backward compatible with the existing Web infrastructure, it inherits all its benefits. First of all, web browsers support the technology natively, including an origin-based security model. The Web infrastructure provides URL-based endpoints, which allows to run multiple services on a single TCP port. Finally, it enables the traversal of proxies and firewalls.

The WebSocket protocol remains a fully functional protocol that can be used outside the browser.

3.1 The WebSocket header

The WebSocket frame structure is shown in Figure 2.

The WebSocket frame structure is composed of different parts [5] [6] :

FIN indicates whether the frame is the final fragment of a message.

Opcode indicates the type of the payload data: binary, textual, or protocol-level signaling (e.g. `close`, `ping`, `pong`).

Mask indicates whether the payload data is obfuscated (only for messages sent from the client to the server).

Length indicates the payload length. If this value lies between 0 and 125, this is the actual length. If it is set to 126, the next 2 bytes represent a 16-bit unsigned integer indicating the frame length. If it is set to 127, the next 8 bytes represent a 64-bit unsigned integer indicating the frame length.

Masking key is used to disable unwanted payload processed by network intermediaries, like HTTP proxies [6]. This part is omitted in server-originated frames, as the masking is not necessary. See section 5.2 for more information.

Payload corresponds to the application data, or custom data if the client and server use a protocol extension.

3.2 The WebSocket handshake

The WebSocket protocol is an application-level protocol built on top of TCP [4]. It enables bidirectional data transport in Web sessions. The WebSocket protocol can be used as a Web-based, near real-time communication channel.

As the WebSocket is a TCP-based protocol, it needs a TCP connection to be established between the client and the server before any WebSocket-based interaction can be performed. As shown on figure 3, the first step is to create a TCP connection, whose handshaking needs three messages between the client and the server (three-way handshake). At this point, both have access to the plain TCP protocol and they can send application-specific data to each other. As it is designed to cope with existing HTTP infrastructure, the client is going to use HTTP to negotiate a switch from

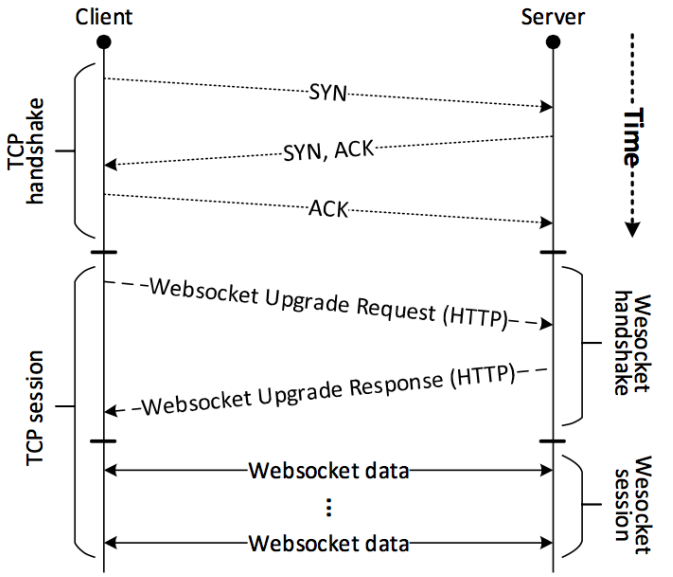

Figure 3. WebSocket-over-TCP sequence diagram (from [6])

regular HTTP to the WebSocket protocol for the rest of the session. To do this, the WebSocket protocol uses the HTTP/1.1 Upgrade header in the request message. This header was designed to allow a client to upgrade from an insecure connection to a secure TLS connection. WebSocket has extended the HTTP Upgrade flow with custom WebSocket headers to perform the negotiation [5]. The client also negotiates a version of the WebSocket protocol to be used for the session. If the server supports the WebSocket protocol, it replies with an HTTP response using the `101 Switching Protocols` status code. At that point, the HTTP-based communication is finished. The connection can then be used as a two-way communication channel where the client and the server can exchange WebSocket messages.

4 Performance of WebSocket

As we saw in section 3.2, establishing a WebSocket session requires at least five messages. The cost of WebSocket connection establishment was evaluated to be 3.7 times longer than a TCP connection [6].

As observed in section 2, XHR and XHR long polling were the two most common methods to simulate a full-duplex connection. To determine whether WebSocket outperforms these, we are going to inspect the latency and the throughput of each of these techniques. Moreover, we are going to see how good WebSocket performs compared to TCP.

4.1 Comparing the latency

The experimentation compares WebSocket, XHR and XHR long polling with different latencies in the underlying link.

In [7], they imagine an experiment to measure the latency induced by the use of WebSocket compared to XHR and XHR long polling. The server receives weather information every 250 ms from a wind sensor and serves it via several ways (WS, XHR and XHR LP). To compare the latency, clients and servers are synchronized.

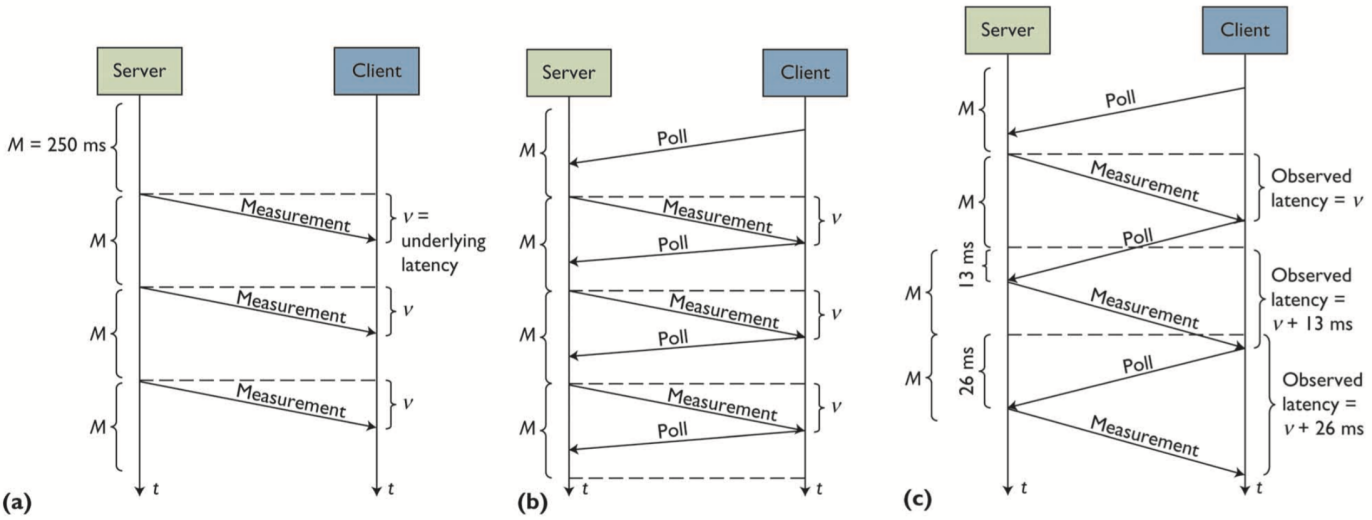

Figure 4. Communication behavior in three situations. (a) is using WebSocket; (b) is using long polling with a low underlying latency; (c) is using long polling with a latency greater than half the data measurement rate. Measurements occur at a constant rate of one every M ms. (from [7])

In this experiment, the XHR without long polling always achieves higher latency. This is understandable given the fact the client has to periodically poll the server for new data. The comparison with XHR is not helpful and will not be considered.

Figure 4, from [7], presents three interesting communication behaviors. Figure 4a illustrates the WebSocket protocol behavior once it has successfully established the socket connection. Each time the server receives a measurement from the sensor, the server can immediately send the measurement to the client. Figure 4b illustrates the XHR long polling behavior with an underlying latency low enough ($<$ 125ms, which corresponds to half the 250ms observation rate) for the client to establish a poll request that the server will keep until it receives a new measurement from the sensor. Figure 4c illustrates the XHR long polling behavior with an underlying latency higher than 125ms. The client first polls the server, the request arrives at the server before or at the exact time a measurement is available. When a measurement is available, the server sends the message immediately from the sensor to the client, which closes the connection.Thus, when the server receives a new poll request from the client, at least one sensor is in the client's queue. The observed latency is greater than the underlying link. The long polling is a viable situation as long as the underlying latency is less than half the data measurement rate. In situations (a) and (b) of the figure 4, the observed latency is the underlying latency, meaning that XHR long polling can perform as good as WebSocket.

Another experiment in [2] is done over a simulated satellite link. They set up a dummy network introducing such a latency that the round trip time is fixed at 600ms. Then the server or the client has to send 100 messages of 39 bytes. The results are similar. For WebSocket the average latency is of about 600ms, the latency of the dummy network. And for the XHR long polling, the average latency is about 1200ms, twice the delay introduced. This makes sense as every time the client (resp. server) has to send (resp. receive) a message in XHR long polling, a new connection has to be established before the data can be issued. The standard deviation of these values is low. This is explained by the fact that the whole experiment was conducted on a small network without congestion.

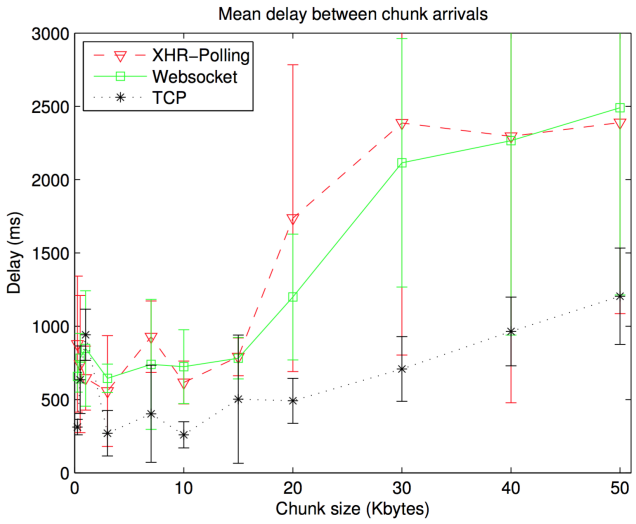

Figure 5. Delay over 3G cellular data network (from [8])

These experiments show that WebSocket achieves better performance than XHR. But what about WebSocket versus plain TCP ? This experiment was conducted in [8]. The setup is a web application that is a simple data echo service. Clients send a fixed-size chunk of random data to the server that simply sends it back to its respective client. Figure 5 shows the delays in receiving consecutive data chunks as a function of data chunk size. This experiment is conducted on a 3G cellular data network with three different protocols : XHR long polling, WebSocket and TCP. We are not going to rediscuss the results concerning WebSocket versus XHR long polling. We can see that which other tests highlighted remains valid in this experiment. The delay is increased by a factor of 2-3 when using WebSocket as compared to

raw TCP. Authors of this study conjectured that the reasons for these additional delays are due to additional buffering time, more chatty protocols, and additional overhead. The first point is mentioned in the IETF HTML5 WebSocket specification "the implementation MAY delay the actual transmission arbitrarily, e.g., buffering data so as to send fewer IP packets." [4]. Also, we can notice that the standard deviation is larger for WebSocket compared to TCP. This standard deviation is called the jitter measurement in networking. Authors of [8] made another experiment to measure the jitter of each of these protocols.

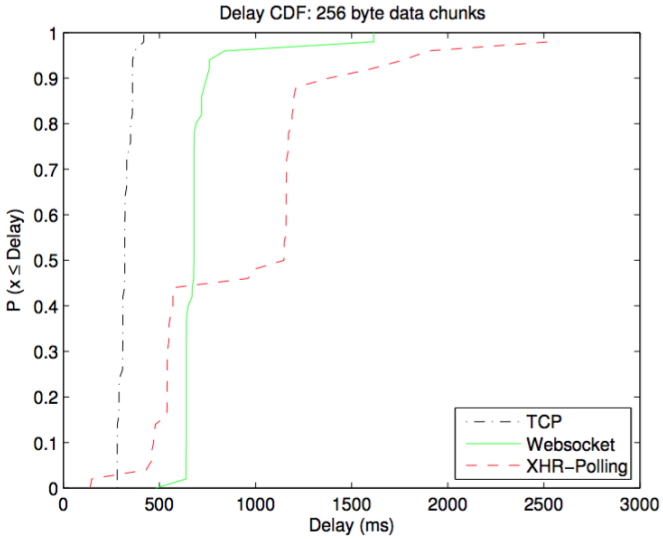

Figure 6. Cumulative distribution function of delays between 256-byte chunk arrivals (from [8])

To do this, they measured the delay between successive 256-byte chunks. The results are shown in figure 6, in a cumulative distribution function. The figure shows that, with TCP, we can expect that over 90% of all data chunks will arrive at the client with an inter-chunk delay of 360ms or less. The corresponding numbers for WebSocket and XHR long polling are 760ms and 1382ms respectively. There is a performance gap between TCP and WebSocket protocol. This phenomenon is accentuated by the fact that the size of the chunks is small. As the transmission delay is short for small chunks, the main factor of the latency depends on the underlying latency and the processing time of protocols. To better understand the reason of this gap, we must analyze the generated overhead.

4.2 Comparing overhead and throughput

Authors of [8] also studied the throughput of the WebSocket compared to TCP and XHR long polling. To do so, they used a new setup with a dummy network. This dummy network will be used to create a traffic-shaped network which restricts total egress bandwidth to 512kbit/sec and introduces a fixed latency of 50ms. It will also capture packets to compute the overhead. The rest of the setup is the same as before, the size of the chunks is fixed and the web service is a simple echoing. In figure 7, they have measured the overhead induced by varying the chunks size. As we can see, the overhead for XHR long polling is very important.

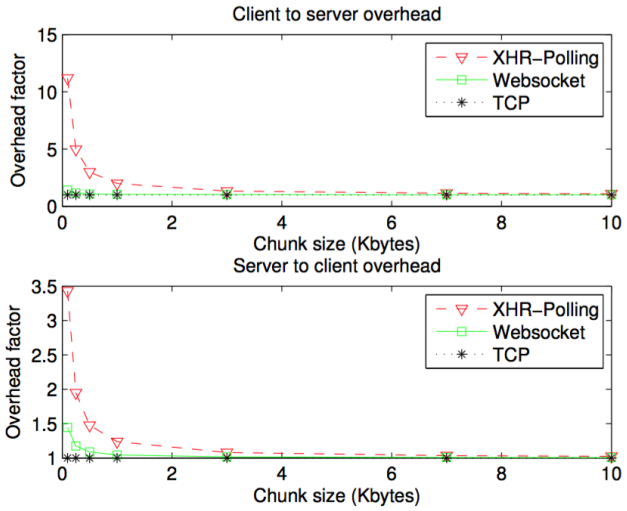

Figure 7. Overhead over a traffic-shaped network (from [8])

For small chunks, it can be up to ten times the size of the data. This is easy to understand as XHR uses HTTP. Thus, the HTTP header is included in each request and response in XHR long polling. And the HTTP header size is unpredictable as it contains a lot of superfluous information, such as browser cookies. It explains why the overhead from the client to the server is more important than the overhead from the server to the client as the client is more likely to add unnecessary headers. We can see that the overhead factor for WebSocket is more reasonable. This overhead is due to the header of the WebSocket protocol described in section 3.1. For a chunk of 256 bytes transmitted from the client to the server, the XHR long polling imposes a large overhead factor of up to 5 times the size of the chunk, Whereas the overhead factor of WebSocket is at 1.16.

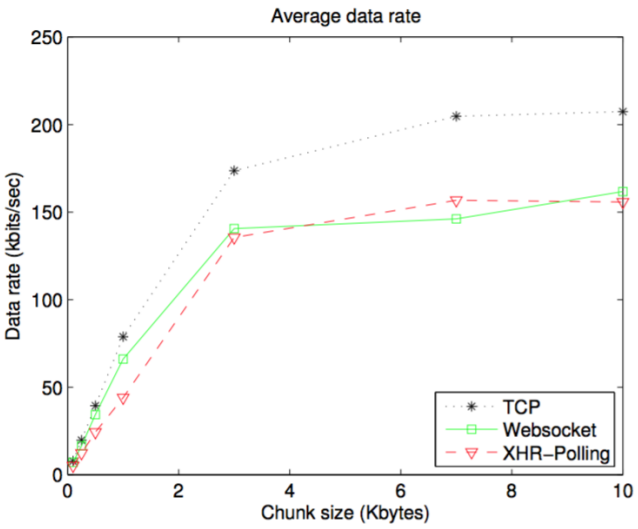

Figure 8. Throughput (from [8])

Authors of [8] also studied the throughput of the WebSocket compared to TCP and XHR long polling. To do so, they used the same setup as described before (traffic-shaped network). The figure 8 shows the net throughput (amount of

useful data received, stripped of protocol overhead, in kbits/s) at the client as a function of data chunk sizes. We can see that TCP performs better throughput than WebSocket and XHR long polling. For a 1kB chunk, TCP achieves a throughput of 78kbits/s as compared to 66kbits and 44 kbits/s in the case of WebSocket and XHR long polling respectively. We know that the throughput is dependent on the round trip time and the overhead, so these results are not surprising.

In [9], authors also analyzed the performance of WebSocket throughput for electric vehicles communicating with WebSocket. They have created an Android application simulating an electric vehicle. The application uses the 3G cellular data network to communicate with a server. They conclude that WebSocket is inefficient in comparison to regular TCP as TCP requires 40%-50% less bit rate than WebSocket for their messages. The size of the status messages of their electric vehicles is of about 500 bytes. As the message is shorter, the overhead ratio is higher. These results are in agreement with the previous ones. The authors highlight the fact that WebSocket may require keepalive traffic to keep the connection open due to the policy of some proxies. These keepalive messages can use a lot of unnecessary traffic, especially if the amount of data exchanged is low.

5 Security

As mentioned in section 3, WebSocket benefits from the HTTP's security model. Also, using existing HTTP infrastructures causes some problems.

5.1 Same origin security

During the handshake, the browser adds the `Origin` header to the HTTP Upgrade request. The server can accept or deny requests based on this header. Also, if they do not share the same origin, it is possible to establish a connection thanks to the Cross Origin Request Sharing (CORS) [10]. The principle is the same as in HTTP.

5.2 Proxies and black boxes

As the web is the most common way to communicate on the internet, there are a lot of proxies and black boxes that intercept the traffic going through port 80. These intermediaries might not understand what happens once the negotiation for WebSocket Upgrade is achieved and this can be problematic. When a proxy or black box does not understand what has happened, the traffic can be altered or discarded. This will make the WebSocket communication unusable. This triggers a problem for the adoption of the WebSockets.

But the most undesired behavior happens when a proxy caches the WebSocket frames. As described in [10], if a proxy is doing such a thing, an attacker can poison the cache of the proxy with malicious payload. When a legitimate user behind the proxy wants to retrieve the WebSocket, he will get the malicious payload from the poisoned proxy's cache. To counter this, a masking key is sent by the client within the header of WebSocket, this key will be used by the server to mask the payload by applying a xor operation between the masking key and the payload. As the client knows the masking key, it can decode the payload and be sure that the payload was not modified by any proxies or black boxes. This mechanism defeats the cache poisoning attack.

5.3 TLS

WebSocket supports TLS. This is useful to cipher payload but not only as proxies and black boxes can do nothing with TLS communication, by ensuring that a website is using secure WebSocket, it also ensures that no proxy or black box will prevent the connection.

6 Conclusion

In this paper, we covered the different solutions that were used prior to the WebSocket. These include XHR polling and XHR long-polling. Then, we introduced the WebSocket, which is composed of the WebSocket JavaScript API and the underlying WebSocket protocol. We saw that the WebSocket protocol was based on TCP. We presented how two hosts could connect to each other to create a WebSocket session. We also exposed the WebSocket header to explain the WebSocket frame structure. We ended the paper by comparing the technology with the prior solutions, with a focus on the throughput and the latency. It turns out that the WebSocket causes lower performance compared to the plain TCP protocol. It causes some overhead and has a higher latency than a plain TCP connection. However, it is still better than the alternative solutions, like XHR polling and XHR long-polling. The overhead caused by the WebSocket is negligible for practical use, and for situations where a lot of data is sent, the initial HTTP handshake is amortized by the large number of payload transfers.

From a security point of view, we saw the threats that exist and how WebSocket counters these. We also explained the importance of the masking key in the WebSocket frames sent by the clients.

Finally, we can conclude this paper by stating that the WebSocket protocol is a useful addition to the Web, which brings a full-duplex communication that is more efficient than the other solutions which try to simulate an equivalent behavior in HTTP-based applications.

References

[1] D. Puranik, D. Feiock, and J. Hill, "Real-time monitoring using ajax and websockets," in *Engineering of Computer Based Systems (ECBS), 2013 20th IEEE International Conference and Workshops on the*, April 2013, pp. 110–118.

[2] M. Collina, A. Vanelli-Coralli, C. Caini, G. Corazza, and R. Secchi, "Latency analysis of real-time web protocols over a satellite link."

[3] I. Grigorik. (2013) High-Performance Browser Networking - XMLHttpRequest. [Online]. Available: http://chimera.labs.oreilly.com/books/1230000000545/ch15.html

[4] I. Fette and A. Melnikov, "The WebSocket Protocol," RFC 6455 (Proposed Standard), Internet Engineering Task Force, Dec. 2011. [Online]. Available: http://www.ietf.org/rfc/rfc6455.txt

[5] I. Grigorik. (2013) High-Performance Browser Networking - WebSocket. [Online]. Available: http://chimera.labs.oreilly.com/books/1230000000545/ch17.html

[6] D. Skvorc, M. Horvat, and S. Srbljic, "Performance Evaluation of Websocket Protocol for Implementation of Full-Duplex Web Streams."

[7] V. Pimentel and B. Nickerson, "Communicating and displaying real-time data with websocket," *Internet Computing, IEEE*, vol. 16, no. 4, pp. 45–53, July 2012.

[8] S. Agarwal, "Real-time web application roadblock: Performance penalty of html sockets," in *Communications (ICC), 2012 IEEE International Conference on*, June 2012, pp. 1225–1229.
[9] J. Perez and J. Nurminen, "Electric vehicles communicating with websockets - measurements and estimations," in *Innovative Smart Grid Technologies Europe (ISGT EUROPE), 2013 4th IEEE/PES*, Oct 2013, pp. 1–5.
[10] L. shung Huang, E. Y. Chen, A. Barth, E. Rescorla, and C. Jackson, "Talking to yourself for fun and profit."