Performance evaluation of Websocket protocol for implementation of full-duplex web streams

Conference Paper · May 2014

DOI: 10.1109/MIPRO.2014.6859715

3 authors:

Dejan Skvorc
University of Zagreb
18 PUBLICATIONS **74** CITATIONS
SEE PROFILE

Matija Horvat
University of Zagreb
3 PUBLICATIONS **26** CITATIONS
SEE PROFILE

Sinisa Srbljic
Intel
64 PUBLICATIONS **495** CITATIONS
SEE PROFILE

Performance Evaluation of Websocket Protocol for Implementation of Full-Duplex Web Streams

D. Skvorc, M. Horvat, and S. Srbljic
University of Zagreb/School of Electrical Engineering and Computing, Zagreb, Croatia
dejan.skvorc@fer.hr, matijah@acm.org, sinisa.srbljic@fer.hr

Abstract – Besides traditional synchronous HTTP request-response communication paradigm that was used on the Web for decades, modern Web services require more flexible communication system that enables asynchronous messaging between clients and servers, as well as server-initiated data delivery. Among several Web-based asynchronous communication paradigms emerged recently, the *Websocket* protocol and corresponding *Websocket* API are accepted as a pivotal framework for implementation of full-duplex asynchronous Web streams.

In this paper, we evaluate the performance of the *Websocket* protocol with respect to underlying TCP protocol. We compare the two against the latency and amount of generated network traffic. The results show that, except a small overhead imposed due to initial handshaking, *Websocket*-based communication does not consume any more network traffic than plain TCP based communication. However, it is still slightly inferior in terms of latency.

I. INTRODUCTION

Since its announcement in the early 1990's, the HTTP protocol is the primary communication protocol for the World Wide Web [1, 2]. HTTP is an application-level protocol built on top of the TCP/IP protocol stack. It was designed to operate in client-server settings with strict separation of concerns between clients and servers. Client is responsible to ask for data or service from a server by sending an HTTP request. On the other hand, server responds with an HTTP response, carrying the results of execution of the action required by the client. Server hosted data and services are delivered to the clients upon their requests only, rather than being delivered autonomously when they become available on the server. Request-response paradigm of the HTTP protocol dictates a synchronous operation of the communication parties, which means that every HTTP request from a client is followed by an HTTP response sent by server. This prevents or at least diminishes the application of the protocol in modern Web-based environments that require more sophisticated and flexible communication models.

Recent research in the field of real-time Web tried to overcome the limitations of the HTTP protocol by upgrading the natively non-asynchronous communication paradigm and tuning it to operate in asynchronous settings. That resulted in various conceptual models, such as HTTP-based polling and long polling [3], and technologies, such as *Comet* [4] and *BOSH* [5]. However, each of these technologies either provides only a subset of the required functionalities or tradeoffs the system's simplicity, scalability, and performance for the sake of functionality. For example, *Comet* and *BOSH* only simulate the asynchronous communication between client and server by maintaining two TCP connections, one for upstream, and another one for downstream communication. Recently introduced *Websocket* protocol [6, 7] and *Websocket* API [6, 8], developed as part of the HTML 5 standard [9], are currently the only true asynchronous full-duplex communication framework for the Web. *Websocket* protocol and *Websocket* API bring the expressiveness and flexibility of the TCP sockets to the Web applications, where access to the raw TCP sockets is otherwise not allowed. *Websocket* technology is now natively supported in all major Web browsers and Web servers, while *Websocket* API libraries are available for almost any programming language used in Web development [10, 11, 12, 13, 14, 15].

Many discussions and written reports have been published so far comparing the performance of the *Websocket* and related technologies [16, 17, 18]. All of them highlight the *Websocket* as a breakthrough technology for implementation of asynchronous full-duplex real-time Web streams. However, little is known about the performance degradation this high-level protocol imposes over a plain TCP socket based communication, which still remains an underlying transport mechanism.

In this paper, we compare the *Websocket* and the TCP protocol against two performance-related dimensions: latency and amount of generated network traffic. Although both protocols provide similar communication capabilities, in general they should not be considered as alternatives to each other since they operate at different levels of the Internet protocol stack. While native applications may directly access the raw TCP protocol through the sockets API, until recently Web applications were limited to the HTTP protocol and its synchronous request-response communication paradigm. The *Websocket* protocol is an application-level protocol built on top of the TCP and specifically designed to expose TCP-like communication capabilities to Web applications, which do not have access to the raw TCP sockets. Thus, the goal of our research was to determine the amount of inevitable performance loss if full-duplex asynchronous communication system operates within the limitations of Web settings.

The rest of this paper is organized as follows. Section 2 outlines the most important characteristics of the *Websocket* protocol and shortly describes the *Websocket* API. We emphasize the *Websocket* handshake and *Websocket* frame as key factors used during the evaluation. In Section 3, we describe the evaluation procedure, the experiments for benchmarking the performance of the *Websocket* protocol, and the results gained from these experiments. Section 4 concludes the paper.

II. WEBSOCKET PROTOCOL AND WEBSOCKET API

The *Websocket* protocol [6, 7] is an application-level protocol built on top of the TCP. It enables bidirectional data transport in Web sessions, and is considered as a viable alternative to HTTP polling techniques [3], which are implemented as tradeoffs between functionality, efficiency, and reliability. As such, the *Websocket* protocol is suitable for Web-based near real-time traffic required in games, chats, stock exchange information systems, multimedia systems, remotely controlled systems, etc.

Although similar in functionalities to the regular TCP protocol, the *Websocket* protocol is backward compatible with the existing Web infrastructure. The *Websocket* handshake is compliant with the HTTP 1.1 specification. Furthermore, *Websocket*-based communication inherits all the benefits of the existing Web infrastructure, such as

- native support in Web browsers, including origin-based security model
- proxy and firewall traversal
- URL-based endpoints which enable multiple, and potentially unlimited number of services running on a single TCP port
- elimination of length limits imposed by plain TCP protocol

Fig. 1 shows a sequence diagram of a *Websocket* session and emphasizes the communication overhead the *Websocket* protocol requires over the plain TCP protocol. Since the *Websocket* is a TCP-based protocol, it requires a TCP connection to be established between client and server before any *Websocket*-based interaction can occur. Handshaking a TCP connection requires three messages to be exchanged between client and server. At this point, two peers that have direct access to the plain TCP protocol can start sending application-specific payload data to each other. However, for *Websocket*-based communication, a *Websocket* session should be established first.

To establish a session, client sends a *Websocket Upgrade Request* to the server, upon which server responds with a *Websocket Upgrade Response*. From this point forward, even the Web-based client and server can send data back and forth in asynchronous full-duplex mode.

A. Protocol Handshake

The *Websocket* handshake takes the advantage of the HTTP protocol in the session establishing phase. The *Websocket Upgrade Request* is a regular HTTP GET request with session endpoint specified as a request URI component. HTTP headers `Upgrade` and `Connection` are indication to the server that the client wants to switch from regular HTTP to the *Websocket* protocol for the rest of the session. Other fields enable negotiation of additional *Websocket* options, such as protocol version, subprotocol indication, and security-related parameters, such as script origin and *Websocket* key used to confirm the connection acceptance.

```
GET /chat HTTP/1.1
Host: server.example.com
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Key: dGhlIHNhbXBsZSBub25jZQ==
Origin: http://example.com
Sec-WebSocket-Protocol: chat, superchat
Sec-WebSocket-Version: 13
```

If the server supports the *Websocket* protocol, it replies with an HTTP response with a status code set to `101 Switching Protocols`. From this point forward, the HTTP-based communication is finished and all further communication is based on *Websocket* frames.

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: s3pPLMBiTxaQ9kYGzzhZRbK+xOo=
Sec-WebSocket-Protocol: chat
```

B. Data Framing

Units of data transfer in the *Websocket* session are called *Websocket* frames. The *Websocket* protocol supports binary data frames, textual UTF-8 encoded data frames, and control frames for protocol-level signaling. Minimal framing information is added to each payload data, as shown in Fig. 2.

Each *Websocket* frame begins with at least 2 bytes long header prepended to the payload data. Depending on the length of the payload data and the direction of the communication, the length of the header may increase up to 14 bytes.

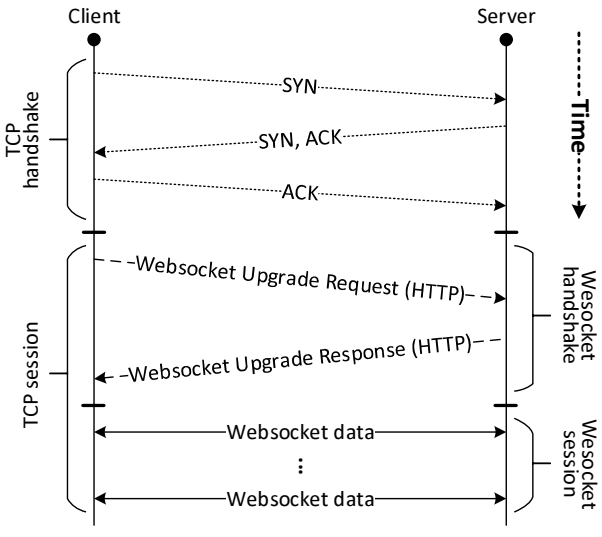


Figure 1. *Websocket*-over-TCP sequence diagram

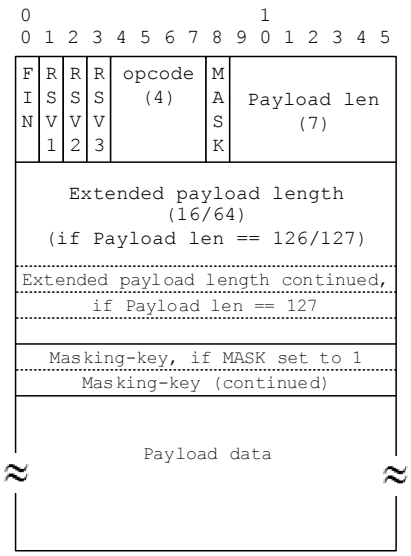


Figure 2. *Websocket* frame structure

First byte contains control bits and 4-bit operation code `opcode`, which defines whether the payload data should be interpreted as binary data, textual data, or protocol-level signaling. Next bit is a `MASK`, which if set to 0 signals that the payload data are transferred in clear form, and obfuscated otherwise. Following 7 bits are used to indicate payload data length. If payload data length is up to 125 bytes, this value is directly specified in `Payload len` field. If payload length is between 126 and 65535 bytes, the `Payload len` field is set to 126, and two additional bytes are added to specify the actual payload data length. Finally, if payload data are longer than 65535 bytes, the `Payload len` field is set to 127, and 8 additional bytes are added to specify the actual payload data length.

Client-originated frames must additionally be obfuscated with a 32-bit `Masking key`, specified in the following 4 bytes of the frame header. This is required to disable the unwanted payload processing performed by the network intermediaries that may discover patterns of HTTP traffic in *Websocket* frames. For example, if a *Websocket* frame contains a pattern of HTTP traffic, an HTTP proxy may accidentally decide to cache the data as if they were regular HTTP traffic. This is a security vulnerability known as HTTP proxy cache poisoning [19, 20]. For frames sent from server to the client, this masking is not necessary, thus 4 bytes containing a `Masking key` are omitted in server-originated frames. The *Websocket* frame ends with payload data of specified length.

Table 1. Event-driven *Websocket* API

Callback	Description
`onopen`	invoked when *Websocket* session is established, signalizes that the protocol is ready to transfer payload data
`onerror`	invoked whenever an error occurs
`onclose`	invoked when one of the peers has terminated the session
`onmessage`	invoked when an incoming message from another peer has arrived

C. *Websocket API*

Besides the protocol itself, *Websocket* specification also defines the API to interact with the protocol [6, 8]. The *Websocket* API is defined by its states of readiness, responses to a networking or messaging event, and message types available for data transfer between client and server. Table 1 summarizes the protocol events exposed to applications through an event-driven callback-based API.

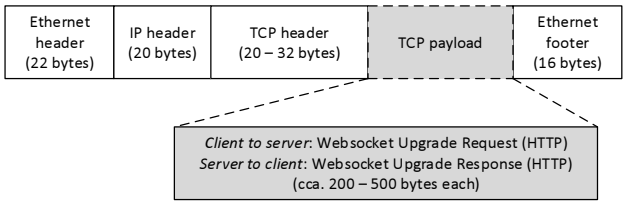
III. PERFORMANCE EVALUATION

Performance evaluation of the *Websocket* and the TCP protocol we present in this paper consists of comparison of the network traffic and data transfer time these two protocols consume to transfer given amount of data between client and server. Since the size of data frames in both protocols depends solely on the size of the payload data, we evaluate the protocols' efficiency from the network traffic perspective analytically, using the protocol specifications defined in *Websocket* RFC [7] and TCP RFC [21], respectively. On the other hand, data transfer time is evaluated experimentally in a laboratory test bed.

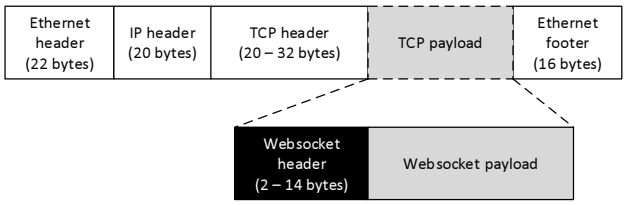
A. *Analitical Evaluation of Network Traffic*

Fig. 3 shows the embedding of the protocol fields in the Internet protocol stack. Regardless of whether the two peers communicate directly via TCP or through the *Websocket* protocol, lower level protocol fields, including the TCP header, are always present in all network packets. Thus, our analysis takes into account only the overhead the *Websocket*-based communication incurs over the plain TCP-based communication. This overhead includes two HTTP messages transferred during the *Websocket* handshake, as well as *Websocket* frame headers for each frame transferred during the session.

The overhead of the *Websocket* handshake is fixed in length and typically counts few hundreds of bytes. Since this handshake is performed once per session, its significance decreases with each new *Websocket* frame carrying payload data. Thus, the rest of our evaluation is focused on long-running sessions, where the impact of the



a) Network packet structure during the *Websocket* handshake



b) Network packet structure during the *Websocket* session

Figure 3. Protocol relationship in the Internet protocol stack

initial handshake is diminished and the only observable traffic overhead is caused by the *Websocket* data framing.

When given amount of payload data are transferred using plain TCP protocol, these data are directly embedded as TCP payload. However, when transferred as a *Websocket* frame, then the TCP payload consists of both payload data and a *Websocket* frame header, as shown in Fig. 3b. This relation is given in (1), where *data* is the length of the payload data, P_{TCP} is the length of the TCP payload when these data are transferred as a TCP payload, P_{WS} is the length of the TCP payload when the same data are transferred as a *Websocket* payload, while H_C is the length of the *Websocket* frame header for client-originated frames.

$$\begin{aligned} P_{TCP} &= data \\ P_{WS} &= data + H_C \end{aligned} \tag{1}$$

As shown in Fig. 2 and described in Section 2, the *Websocket* protocol uses variable frame header length that depends on the size of the payload data. The *Websocket* framing scheme is given in (2), where *data* is the length of the payload data, H_S is the length of the frame header for server-originated frames, while H_C is the length of the frame header for client-originated frames. As described in Section 2, masking of client-originated frames requires 4 additional header bytes in relation to server-originated frames.

$$\begin{aligned} H_S &= \begin{cases} 2, & 0 \le data \le 125 \\ 4, & 126 \le data \le 65535 \\ 10, & data \ge 65536 \end{cases} \\ H_C &= H_S + 4 \end{aligned} \tag{2}$$

The network traffic overhead O_P a *Websocket*-based communication incurs over a plain TCP-based communication is given in (3). The dependence of this overhead on the size of the payload data is given in Fig. 4. The graph shows a worse case, with a client-to-server directed communication only, where each *Websocket* frame is prepended with 4 additional bytes carrying a masking key.

$$O_P = \frac{P_{WS} - P_{TCP}}{P_{TCP}} \cdot 100\% \tag{3}$$

Significant difference in performance of the two protocols occurs only for tiny messages of just few bytes in size. In this range, the *Websocket* protocol generates almost an order of magnitude more network traffic than the TCP. However, even with this additional traffic, the total overhead is no more than six bytes, as defined in (2). For bigger messages, the *Websocket* frame size converges very fast towards the non-framed TCP payload size. For messages bigger than 1 kB, the overhead is less than 1 %. The relation shown graphically is given for small range of payload data sizes only, where the difference in the behavior of the two protocols is significant. We intentionally omitted the rest of the graph, where this relation asymptotically approaches to 0.

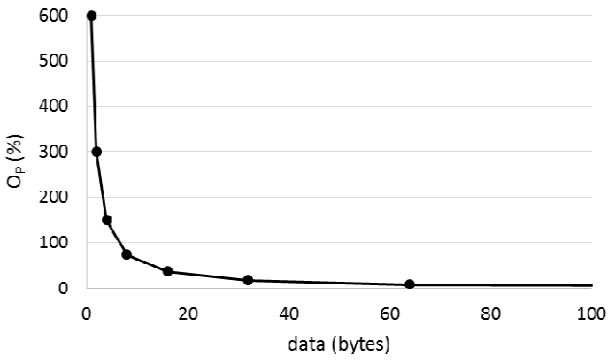


Figure 4. Network traffic overhead of the *Websocket* protocol in relation to the TCP protocol

Except the initial handshaking performed through one HTTP round trip at the beginning of the session, the amount of network traffic generated by the *Websocket* protocol is comparable to that generated by the plain TCP. With a carefully designed framing scheme, which gradually decreases the amount of framing information as the payload data decrease in size, the *Websocket* protocol generates negligibly more network traffic than plain TCP, even for the smallest amounts of data.

B. Experimental Evaluation of Data Transfer Time

Experimental environment for measuring data transfer time consists of two host machines, one playing the role of a server, while other being a client. Hardware and software configuration of these host machines are given in Table 2.

Both protocols are similar in terms that they both require particular form of negotiating the protocol parameters between communicating parties, prior to be able to transfer actual data through the network. Thus, we separately measure the time required to negotiate a connection and the time required to transfer a chunk of payload data from one side to the other.

In our first experiment, we measure the time required for client and server to establish a TCP connection and a *Websocket* session, respectively. Results are given in Fig. 5.

Table 2. Experimental environment configuration

	Client	**Server**
Hardware	CPU: AMD Turion II P520 RAM: 6 GB	CPU: AMD Athlon X2 5000+ RAM: 5 GB
OS	Windows 8 64-bit	Windows 8 64-bit
Network	1000BASE-T (Gigabit Ethernet, host machines directly connected using UTP Cat5 Ethernet cable)	
TCP implementation	*java.net.Socket* (Java JDK 1.7)	*java.net.Socket* (Java JDK 1.7)
Websocket implementation	*org.eclipse.jetty.websocket.**client**.** (Jetty 9.1.0)	*org.eclipse.jetty.websocket.**servlet**.** (Jetty 9.1.0)

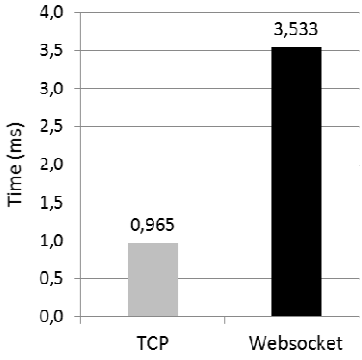


Figure 5. Comparison of TCP and *Websocket* handshake duration

In our laboratory environment with client and server being directly connected with a Gigabit Ethernet cable, establishing a *Websocket* session lasts 3.7 times longer than establishing a TCP connection. The reason for such a slow performance of the *Websocket* protocol is the fact that the *Websocket* protocol is not a real transport protocol, but rather an application-level protocol built on top of the TCP. This means that each *Websocket* session requires a TCP connection to be established first, as shown in Fig. 1. Thus, setting up a *Websocket* session includes all the actions required for setting up a TCP connection, plus an additional HTTP round trip for a *Websocket* handshake. Besides the time required to transfer two HTTP messages through the network, this additional handshake also includes the time necessary for client and server to allocate the necessary resources and initialize the *Websocket* API callbacks required to perform a *Websocket*-based communication in the rest of the session.

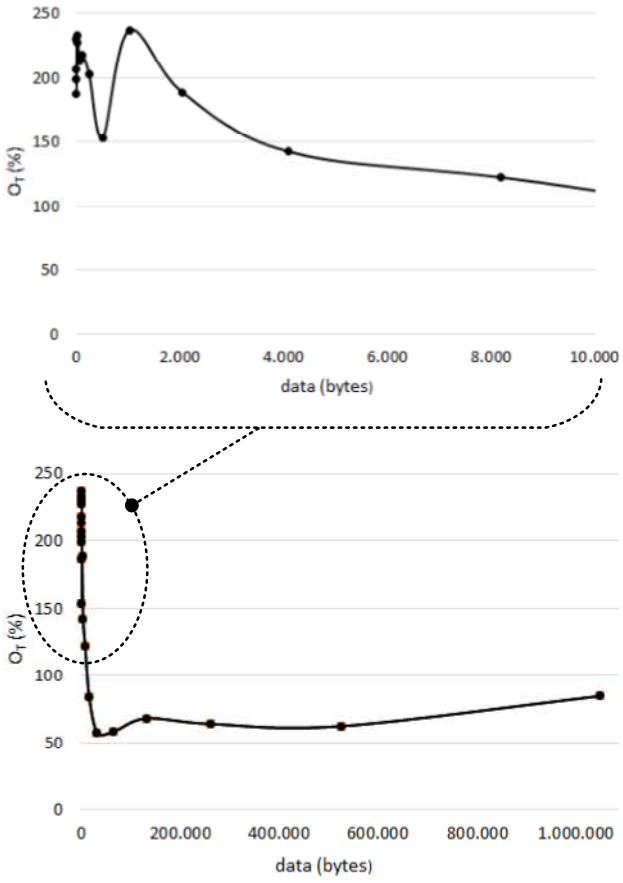


Figure 6. Data transfer time overhead of the *Websocket* protocol in relation to the TCP protocol

In our second experiment, we measure the data transfer time between client and server once the TCP connection and the *Websocket* session have been established. Client generates a given amount of payload data and sends them to the server. Upon receiving all the data, server echoes the same data back to the client. Client measures a round trip time. Results are given in Fig. 6. On vertical axis, we show a relative time overhead O_T a *Websocket*-based communication incurs over a plain TCP-based communication. This overhead is calculated by (4), where T_{TCP} is the time required to transfer the data using plain TCP protocol, while T_{WS} the time required to transfer the same amount of data using the *Websocket* protocol.

$$O_T = \frac{T_{WS} - T_{TCP}}{T_{TCP}} \cdot 100\% \quad (4)$$

Here we notice that, despite of the negligible difference in the overall amount of data being transferred (payload + *Websocket* frame headers), the *Websocket* protocol performs significantly slower than the TCP through the whole spectrum of the test cases. The performance drop is more significant for small messages and gets better as message size increases. The overhead of the *Websocket*-based communication fluctuates between 150 % and 250 % for very small messages. For bigger messages, as message size becomes more and more dominant factor in data processing and data transfer, the overhead gets stable at roughly 60-70 %.

There are several factors that impact the performance of the *Websocket* protocol. First reason is the relationship of the TCP and the *Websocket* protocols in the Internet protocol stack. The *Websocket* operates at a higher level of the stack and consumes more computational resources that deliver the data from the TCP layer to the upper layers. Second, opposed to TCP, the *Websocket* protocol uses an event-driven callback-based API to deliver data to Web applications, which requires additional application-level data handling. Third, every message sent from a client to a server requires additional processing to mask the data with a 32-bit masking key on a client, and to unmask them on a server. One more reason, yet unconfirmed in our experiments, might be the implementation of the protocol itself. While TCP protocol is intensively used for decades and highly optimized libraries are available for almost any programming language, the *Websocket* protocol emerged just a few years ago, with little production systems deployed so far, and with early implementations not yet verified by wide research and developer community.

IV. CONLUSION

In this paper, we have analytically and experimentally compared the *Websocket* and the TCP protocol against two performance-related dimensions: amount of generated

network traffic and data transfer time. As expected, TCP protocol slightly outperforms the *Websocket* protocol in both of these dimensions because the *Websocket* operates at the application layer and uses the underlying TCP as a transport layer protocol. While the difference in the amount of generated network traffic is minimal and negligible for any practical use, the degradation in data transfer time is still remarkable due to operation at a higher level of the Internet protocol stack, data masking, and event-driven callback-based interface to applications. However, the advantage of the *Websocket* protocol is its alignment with the existing Web infrastructure, where low-level TCP protocol is not directly applicable.

Since the *Websocket* protocol augments the Web with a TCP socket-like communication capabilities that were not available or were only inefficiently simulated in HTTP-based applications, slight drop in performance in relation to the plain TCP protocol still makes it a powerful mechanism for implementation of full-duplex asynchronous Web-based data streams. If used as intended, in long-running Web sessions, where data are continuously streaming between client and server, the cost of initial HTTP handshaking is split over the large number of optimized payload transfers and gets negligible even after few client-server interactions.

ACKNOWLEDGMENT

This work is sponsored by the Ministry of Science, Education, and Sports of the Republic of Croatia under the research grant agreement 036-0362980-1921.

REFERENCES

[1] D. Gourley et al., HTTP: The Definitive Guide. Sebastopol, CA: O'Reilly Media, Inc., 2002.

[2] R. Fielding et al. (1999, June). *Hypertext Transfer Protocol – HTTP/1.1* [Online]. Available: http://tools.ietf.org/ search/rfc2616

[3] S. Loreto, P. Saint-Andre, S. Salsano, and G. Wilkins. (2011, April). *Known Issues and Best Practices for the Use of Long Polling and Streaming in Bidirectional HTTP* [Online]. Available: http://tools.ietf.org/search/rfc6202

[4] P. McCarthy and D. Crane, Comet and Reverse Ajax: The Next-Generation Ajax 2.0. New York, NY: Apress, 2008.

[5] I. Paterson, D. Smith, P. Saint-Andre, and J. Moffitt. (2010, July). *XEP-0124: Bidirectional-streams Over Synchronous HTTP (BOSH)* [Online]. Available: http://www.xmpp.org/extensions/ xep-0124.html

[6] V. Wang, F. Salim, and P. Moskovits, The Definitive Guide to HTML5 Websocket. New York, NY: Apress, 2013.

[7] I. Fette and A. Melnikov. (2011, December). *The Websocket Protocol* [Online]. Available: http://tools.ietf.org/search /rfc6455

[8] I. Hickson. (2012, September). *The Websocket API* [Online]. Available: http://www.w3.org/TR/websockets/

[9] R. Berjon et al. (2014, February). *HTML5: A Vocabulary and Associated APIs for HTML and XHTML* [Online]. Available: http://www.w3.org/TR/html5

[10] *Python Websocket API* [Online]. Available: https://pypi.python.org/pypi /websocket/0.2.1

[11] J. Vos. (2013, April). *JSR 356 Java API for Websocket* [Online]. Available: http://www.oracle.com/technetwork/articles/ java/jsr356-1937161.html

[12] *C++ Websocket API* [Online]. Available: http://www.zaphoyd.com/websocketpp

[13] *.NET Websocket API* [Online]. Available: http://msdn.microsoft.com/en-us/library/system.net.websockets. websocket(v=vs.110).aspx

[14] *Jetty* [Online]. Available: http://www.eclipse.org/jetty/

[15] *Autobahn* [Online]. Available: http://autobahn.ws/

[16] P. Lubbers and F. Grecco. *HTML5 Web Sockets: A Quantum Leap in Scalability for the Web* [Online]. Available: http://www.websocket.org/quantum.html

[17] M. Laine and K. Saila, "Performance evaluation of XMPP on the Web," Finland, 2012.

[18] A. Almasi and Y. Kuma, "Evaluation of Websocket communication in enterprise architecture," Sweden, 2013.

[19] L. Huang, E. Y. Chen, A. Barthy, E. Rescorlaz, and C. Jackson, "Talking to yourself for fun and profit," in Web 2.0 Security and Privacy 2011, Oakland, CA, USA, 2011.

[20] A. Herzberg and H. Shulman, "Socket overloading for fun and cache-poisoning," in Annual Computer Security Applications Conference 2013, New Orleans, LA, USA, 2013.

[21] (1981, September). *Transmission Control Protocol* [Online]. Available: http://www.ietf.org/rfc/rfc793