Performance Evaluation of XMPP on the Web

Markku Laine
48605D
markku.laine@aalto.fi

Kalle Säilä
64775E
kalle.saila@aalto.fi

April 20, 2012

Abstract: In recent years, the trend to move desktop applications to the Web has posed new requirements for Web applications. One of those requirements is real-time communication, which is an essential feature in a variety of different application domains, such as collaborative working, social networking as well as audio and video conferencing applications. HyperText Transfer Protocol (HTTP), the standard data communication protocol on the Web, however, has not been designed with either real-time communication or uni-/multicasting in mind. This in turn has forced developers to resort to use inefficient and error-prone communication techniques. In this paper, we show how Extensible Messaging and Presence Protocol (XMPP) can be used to overcome the problems of real-time communication and uni-/multicasting on the Web. We carried out a study to assess the performance of three different communication techniques, including XMPP over HTML5 WebSocket, in a Local Area Network (LAN) in order to determine which technique is best suited for using the protocol on the Web. Our studies show that XMPP offers an efficient and compelling option for developing Web applications with real-time communication and uni-/multicasting requirements, especially when used over WebSocket.

Keywords: WWW, Real-Time Communication, XMPP, Performace, WebSocket

1 INTRODUCTION

During the past two decades, the World Wide Web (Web) has evolved from a simple document sharing system to a platform for software applications [1].

Many popular desktop applications including office suites (e.g., Microsoft Office, `http://office.microsoft.com/en-us/web-apps/`) and multi-player games (e.g., Quake, `http://www.quakelive.com/`) are already available through the Web. The main reason for this transition is that the Web as a software platform offers numerous benefits compared to conventional binary end-user software, such as instant worldwide deployment; no manual installations or upgrades; platform independence; and ubiquitous, seamless access to data [2].

In addition to the evolution of the Web itself, also the way we interact with Web applications has fundamentally changed in recent years. In the past, a client initiated almost all communications between the client and the server. For keeping the client up-to-date, update requests were sent to the server at predefined intervals. Today, we expect the server to push updates to the client whenever they occur on the server. The problem with implementing server push is, however, that the *HyperText Transfer Protocol (HTTP)* does not properly support it nor does the Web provide any other standardized solution for addressing the issue[1].

In this paper, we focus on *Extensible Messaging and Presence Protocol (XMPP)* [3], which provides a rich set of features and is a widely used open standard for real-time communication outside the Web. We describe the features of XMPP in detail and compare the performance of three different techniques in a Local Area Network (LAN) that allow the use of the protocol on the Web: (1) Ajax [4] with polling over HTTP, (2) Bidirectional-streams over Synchronous HTTP (BOSH) [5], and (3) WebSocket [6]. Our results show that all the aforementioned techniques make it possible to use XMPP on the Web. Furthermore, the tests suggest that using XMPP sub-protocol for WebSocket results in the best performance, regardless of the message size.

Our main contributions are the experimental design and conducted performance tests for three different techniques in a LAN for supporting real-time communication with XMPP on the Web. Furthermore, our results provide initial guidelines for developers when they are designing applications with real-time communication and uni-/multicasting constraints.

The rest of the paper is organized as follows. The next Section presents a brief review of related work. In Section 3, we provide the fundamentals of XMPP and present three techniques for using it on the Web. Section 4 describes the experimental design and conducted performance tests. The results are presented and discussed in Sections 5 and 6. Finally, Section 7 concludes the paper.

[1]The W3C has two specifications, namely Server-Sent Events and WebSocket, in progress.

2 RELATED WORK

Even though XMPP is over a decade old, its applicability for the Web usage has been scarcely studied. In [7], Pohja has examined how the HTTP-based push technologies of that time can be replaced with the help of an instant messaging protocol, namely XMPP. While the paper provides an extensive comparison between HTTP and XMPP, it lacks a comprehensive performance evaluation and ignores the option of using XMPP sub-protocol for WebSocket. Various techniques for using XMPP on the Web, including XMPP sub-protocol for WebSocket, are described in detail in a more recent publication by Jansen [8].

Griffin and Flanagan [9], on the other hand, have covered both a performance evaluation and the technique of using XMPP sub-protocol for WebSocket in their paper, which seeks to identify a suitable mechanism for delivering asynchronous real-time events to browser-based applications. Unfortunately, their performance evaluation was based solely on visual observations rather than quantitative measurements, and thus, failed to rank the techniques in order of superiority.

Bozdag et al. [10] provide an extremely comprehensive description of how they have designed and conducted their experiments when testing the performance of push and pull techniques for Ajax applications. In addition, they present a distributed test framework called *Chiron* in their follow-up article [11]. Gutwin et al. [12] have also tested the performance of different Ajax push techniques (i.e., Comet) and compared them to WebSocket and a plug-in solution, *Java Applets*, in three different network environments: LAN, MAN, and WAN. Their findings suggest that WebSocket performs the best in all network environments. In addition, they recommend developers to consider the browser as a legitimate vehicle for real-time multi-user systems. We used the aforementioned publications as a basis for designing our experiments and in order to be able to better compare our results to earlier work.

3 BACKGROUND

3.1 XMPP

The *Extensible Messaging and Presence Protocol (XMPP)* [3] is an application-level protocol for exchanging information between any network entities in near real time. The protocol was originally developed under the name *Jabber* within the Jabber open-source community in 1999, and its main purpose was to address the issues with instant messaging (IM) and presence applications of that time [13]. Since then, the core of the Jabber protocol has been revised and formalized by the *Internet Engineering Task Force (IETF)*[2], and published under its current name XMPP in their *Request for Comments (RFC)* series as RFC 6120 [14] and RFC 6121 [15]. In addition to the aforementioned core XMPP specifications, the *XMPP Standards Foundation (XSF)*[3] has developed and published over 300 *XMPP Extension Protocols (XEPs)*[4] to cover a wide variety of application scenarios, such as *XEP-0206: XMPP Over BOSH*, which is an HTTP binding for XMPP communications [16] as well as *XEP-0060: Publish-Subscribe* [17] for multicasting messages and *XEP-0045: Multi-User Chat* [18] for chat rooms with authenticated users, which address the challenge of uni-/multicasting on the Web.

Similar to the Web, also XMPP is based on a decentralized client-server architecture. When an XMPP client wants to start a session with an XMPP server, it opens an XML stream over a long-lived connection (e.g., Transmission Control Protocol, TCP [19]) to the server. Next, the server opens another XML stream to the client, resulting in two XML streams over a single TCP socket, one in each direction. Figure 1 illustrates the XMPP interaction model between the client and the server.

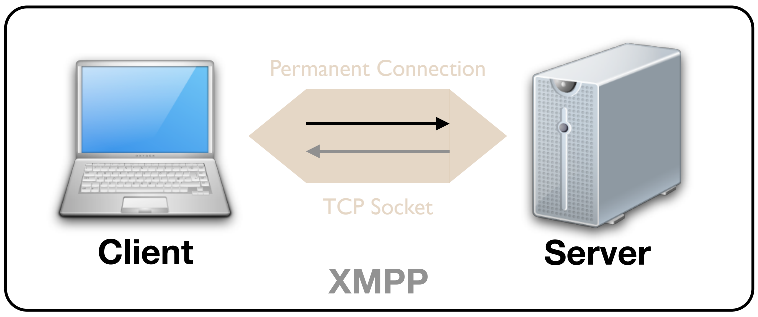

Figure 1: The XMPP-based client/server interaction model.

After the connections have been established, each entity can asynchronously exchange an unbound number of special XML snippets over the streams. These special XML snippets, called *XML stanzas*, define the basic units of communication in XMPP and are as follows: `message`, `presence`, and `iq` (Info/Query). The *Message Stanza*, which is the basis of client to client message transfer in XMPP, consist of a root element called `message` and

[2]IETF, http://www.ietf.org/

[3]XMPP Standards Foundation, http://xmpp.org/about-xmpp/xsf/

[4]XMPP Extension, http://xmpp.org/xmpp-protocols/xmpp-extensions/

a child element called `body`, which is the wrapper for the actual message payload. In addition the root element contains 1-5 attributes, that are: *to* (mandatory), *from*, *id*, *type*, and *xml:lang*. Listings 1 and 2 show examples of an XMPP message with a minimum and maximum amount of attributes.

```
<message to="{username}@{domain}">
  <body>{payload}</body>
</message>
```

Listing 1: Minimal message stanza.

```
<message to="{username}@{domain}/{resource}"
  from="{username}@{domain}/{resource}"
  id="{messageId}"
  type="{chat|error|groupchat|headline|normal}"
  xml:lang="{xmlLang}">
  <body>{payload}</body>
</message>
```

Listing 2: Message stanza with all attributes.

Even though XMPP provides a rich set of features and is a widely used protocol for real-time communication, it is hardly used on the Web. There are two main reasons for this: (1) Web browsers do not provide native XMPP support and (2) regular XMPP communication may be blocked, for example, by firewalls and proxies. The following subsection describes three different techniques that allow the use of XMPP on the Web.

3.2 Techniques for using XMPP on the Web

As stated above, there is no native support for XMPP on the Web because the XMPP protocol does not suite well on the Web architecture as is. HTTP is a stateless protocol based on request/response architecture, whereas XMPP uses stateful, full-duplex interaction channels. However, due to the extensibility of the XMPP protocol, several extensions have been proposed to support XMPP on the Web. The first widely supported and now deprecated extension was *XEP-0025: Jabber HTTP Polling* [20]. Currently, the most popular method for XMPP on the Web is a combination of the extensions *XEP-0124: Bidirectional-streams Over Synchronous HTTP (BOSH)* [5] and *XMPP over BOSH* [16]. The emergence of the WebSocket protocol [6] has also brought a new alternative [7] to support XMPP on the Web.

3.2.1 Jabber HTTP Polling

The XEP-0025: Jabber HTTP Polling [20] was the first XMPP extension protocol that enabled the use of XMPP on the Web with firewalls and proxies.

The basic principle of the extension is that the client makes periodical Ajax requests to the server through the standard HTTP(S) ports. If the server has new data to be delivered to the client, it is sent within the response. Figure 2 demonstrates the Polling-based communication between the client and the server.

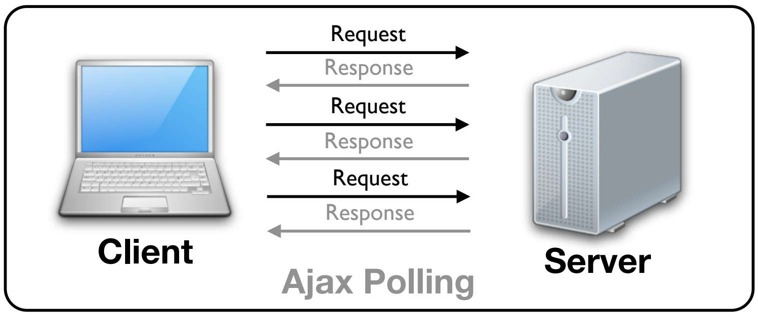

Figure 2: The Polling-based client/server interaction model.

When a Jabber HTTP Polling client sends an XMPP message, the message is prepended with an identifier and a key. Listing 3 demonstrates a *Message Stanza* sent by the client using Jabber HTTP Polling.

```
{identifier};{key},<message to="{username}@{domain}">
  <body>{payload}</body>
</message>
```

Listing 3: Example message stanza with Jabber HTTP Polling.

The Polling technique is an easy way to bring near real-time communication to the Web without the need to use any non-standard mechanism or hacks, but the method is quite inefficient because it generates a lot of unnecessary network traffic between the client and server. Due to the inefficiency of this method, other more efficient methods have emerged to bring real-time communication to the Web. The Jabber HTTP Polling extension is also deprecated in favor of more modern extensions.

3.2.2 Bidirectional-streams Over Synchronous HTTP (BOSH)

Currently, the most popular real-time methods suitable for the Web rely on the existing HTTP standard, but in order to achieve the real-time connectivity some parts of the protocol are being used improperly. These methods, referred as Comet, include *XMLHttpRequest Streaming*, *Inline Frame Streaming*, and *Long-Polling* [8].

XEP-0124: Bidirectional-streams Over Synchronous HTTP (BOSH) [5] with *XMPP over BOSH* [16] is the industry standard for XMPP on the Web. BOSH is based on long-lived HTTP connections (Long-Polling) and is able to offer near real-time connectivity. The basic idea of the protocol is that when a client establishes a connection to the server, the connection is kept open until new data is available or a timeout occurs. As soon as the client receives a response to the request a new connection to the server is established. Figure 3 illustrates the communication between the client and the server over BOSH.

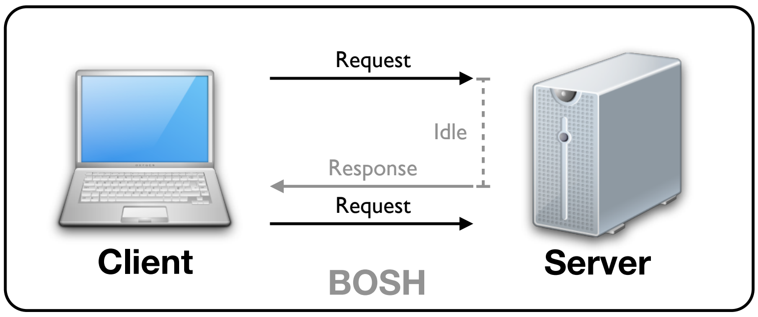

Figure 3: The BOSH-based client/server interaction model.

All *XMPP Stanzas* transmitted through a BOSH connection are wrapped in a new root element called `body`, which indicates the session and the protocol used through its attributes. The wrapper element can contain any number of *XMPP Stanzas* (e.g., multiple new messages that have arrived to the server since the last request/response). Listing 4 shows an example of a server response to a BOSH client.

```
<body rid="{requestId}"
  sid="{sessionIdentifier}"
  xmlns="http://jabber.org/protocol/httpbind"
  key="{key}">
  <message to="{username}@{domain}">
    <body>{payload}</body>
  </message>
  .
  .
  .
  <message to="{username}@{domain}">
    <body>{payload}</body>
  </message>
</body>
```
Listing 4: Example response to a BOSH client.

3.2.3 XMPP over WebSocket

WebSocket [6] is a new standard for real-time communication on the Web. The basic idea of the standard is to provide a full-duplex, bidirectional communication channel over a single TCP socket. The WebSocket standard has many similarities with the XMPP standard on the transport layer, and thus, could be used as a transport mechanism for XMPP on the Web. Figure 4 demonstrates the WebSocket-based communication between the client and the server.

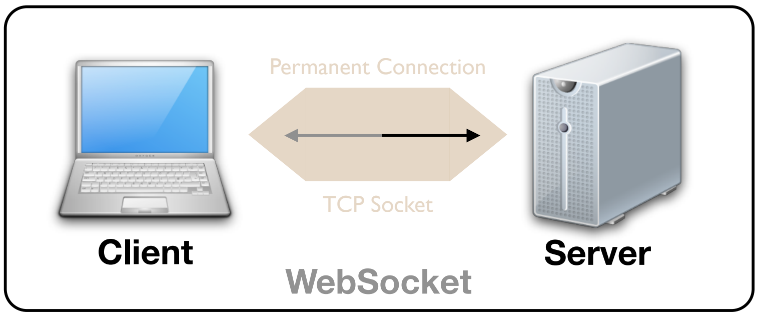

Figure 4: The WebSocket-based client/server interaction model.

Although there is no official XMPP Extension Protocol combining XMPP and WebSocket, all the major XMPP server implementations already have a support for WebSocket-based XMPP communication or the support is coming in the near future. Furthermore, Jack Moffit has submitted an Internet Draft called *An XMPP Sub-protocol for WebSocket* [21] already on December

2010. Based on the fact that WebSocket is the only technique providing true bidirectional, full-duplex communication on the Web, WebSocket will likely become the dominant transport mechanism for XMPP on the Web in the near future. The structure of the *Message Stanzas* sent over WebSocket is similar to the native *Message Stanzas* (see Listings 1 and 2).

4 EXPERIMENTAL DESIGN

The main goal of these experiments was to find out whether XMPP could be justifiably used on the Web and which real-time communication technique should be used on the Web as the underlying communication mechanism. Our aim was to solely test the approaches in terms of performance and network overhead with client applications that did not conduct any other activities at the same time (e.g., drawing to the screen or processing additional data). We also excluded the connection establishment from our tests and focused on the messages with actual payload.

4.1 Setup

To test the different communication techniques we implemented a simple Web application for each of the techniques. The applications were standard HTML documents with a custom JavaScript code utilizing the JSJaC[5] (version 1.3.4) XMPP client library and a simple user interface that was not manipulated in any way during the tests. The applications were otherwise identical except of the connection method (i.e., Jabber HTTP Polling, XMPP over BOSH, and XMPP sub-protocol for WebSocket).

The tests were carried out in a LAN (100/100 Mbit/s) network and the browser used was Safari 5.1.5 on an iMac with a 64-bit OS X 10.6.8 operating system and a 2.4 GHz Intel Core 2 Duo CPU. Our XMPP server was Ejabberd[6] (version 2.1.10) with dedicated modules for handling XMPP communication through our selected techniques, running on a MacBook Pro with a 64-bit OS X 10.6.8 operating system and a 2.53 GHz Intel Core 2 Duo CPU. The WebSocket module used in the Ejabberd supported only WebSocket Draft 03[7] (draft-hixie-thewebsocketprotocol) which is supported only in the Safari browser.

[5]JSJaC, http://blog.jwchat.org/jsjac/

[6]Ejabberd, http://www.process-one.net/en/ejabberd/

[7]WebSocket Draft 03, http://tools.ietf.org/html/draft-ietf-hybi-thewebsocketprotocol-03

We carried out three kinds of test cases that measured:

- Network overhead generated by each request,
- Client-server-client round trip time on varying message sizes, and
- Client message receive rate on varying message sizes.

In each test case for Jabber HTTP Polling, the interval was fixed at 100 ms.

Network Overhead
In this test, we monitored the network traffic between the client and the server to find out the amount of bytes needed to transmit a message from the client to the server. We were especially interested in two things: how much overhead the WebSocket protocol generates compared to the HTTP protocol, and how much additional overhead is created because of the use of XMPP.

Round Trip Rate
Because the core idea of XMPP is to send messages between clients in real-time, we were interested in the time needed to successfully transmit a message from a client to the same client via the server. XMPP servers do not offer any information when the receiving client has actually received the message so we decided to perform a round trip test in which a client sends 100 messages to itself and the client is allowed to send a new message only after the previous message has been retrieved back from the server. We performed these tests with a static payloads of 10, 100, and 1000 bytes messages per approach. Each test set consisted of 30 test runs to gain a sufficient amount of data for statistical analysis.

Message Receive Rate: Server to Client
In addition to the round trip test, we wanted to test the maximum message receive rate from the server to the client in order to simulate a situation in which multiple clients send messages simultaneously to one client. Similar to the round trip test, we conducted a series of tests with varying message payload sizes (i.e., 10, 100, and 1000 bytes) per technique. Each test set consisted of 30 test runs with 10000 identical messages sent from the server to the client.

4.2 Tools

In addition to the test clients and the XMPP server, we used the following tools during testing. Wireshark[8] (version 1.6.2) was used to monitor and analyze the generated network traffic between the client and the server. We also saved all the captures from Wireshark to separate files in order to verify that all the required packets were actually sent and received correctly.

For the message receive rate test, we used a custom, Java-based native XMPP client to generate the 10000 messages to be transmitted to the client application. The Java application ran on the same host machine as the server and we monitored the XMPP traffic to make sure that the send rate of the application was well above the receive rate of our test applications. It would have also been possible to store the 10000 messages to the server prior to establishing a connection to the client (i.e., no need to monitor the send rate), but messages delivered to clients from the offline storage contain additional XML payload (an XML fragment that indicates the original send time of the message), that would have affected the test results because of the additional overhead.

5 RESULTS

Like stated in the previous section, we tested the techniques with test clients not performing any other tasks while sending or receiving messages, meaning that the achieved rates during the tests would be somewhat smaller in a real-world application. Also the 100 ms interval used with Jabber HTTP Polling is too dense to be used in an application performing additional tasks. The results gained from the experiments are presented in the following subsections. Diagrams representing round trip and receive rates are in logarithmic scale and the results are represented in median values.

Network Overhead

Figure 5 represents the theoretical percentage of the overhead in a single packet sent from the client to the server in each technique in relation to the payload sent along the message. The additional overhead consists of the protocol headers and *Message Stanza.* As shown in the figure, the overhead of the Polling and BOSH techniques are quite similar because both of them are HTTP based. The overall network overhead with BOSH is bigger than in Polling because of the additional wrapper XML element required in BOSH *Message Stanzas*, although the size of the HTTP Headers in BOSH is a bit

[8]Wireshark, http://www.wireshark.org/

smaller than in Polling. Without the overhead caused by the *Message Stanza*, WebSocket would have far less network overhead than the HTTP-based techniques. As stated before, the figure only represents single packets sent from the client to the server. If one wants to calculate the overall network overhead caused by the client, one should keep in mind that in HTTP-based techniques each request results to a response which almost doubles the overhead. In addition, the TCP protocol produces some extra traffic since the connection endpoints must send acknowledgment (ACK) messages to each other when receiving packets. Even if the figure suggests that Polling would be more efficient than BOSH in terms of network overhead, in most real-world applications the Polling technique generates a lot of unnecessary requests because the server always sends a response immediately even if there is no new data on the server, and thus the total network overhead of the Polling-based client rapidly increases over the total network overhead generated by a BOSH-based client.

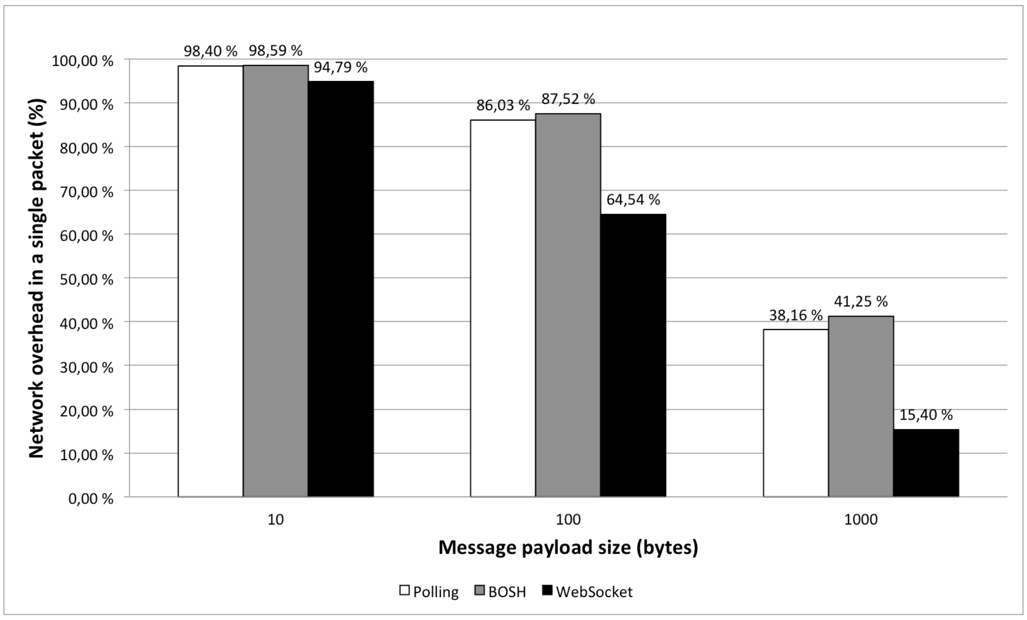

Figure 5: Percentage of network overhead in a single packet.

In Figure 6, we have separated the sections of a packet in each technique to verify the reasons for different network overheads. Because all techniques are TCP/IP-based, they have a 22 bytes Ethernet Header, 20 bytes IPv4 Header, 32 bytes TCP Header (out of which 12 bytes Options), and a 16 bytes Ethernet Footer. The XMPP overhead is quite similar in each packet

(90 bytes in WebSocket, 257 bytes in BOSH, and 162 bytes in Polling) regardless of the message payload size. The real difference is in protocol headers, where BOSH has 355 bytes (HTTP header) and Polling has 365 bytes (HTTP header), whereas WebSocket only adds 2 bytes to the total overhead.

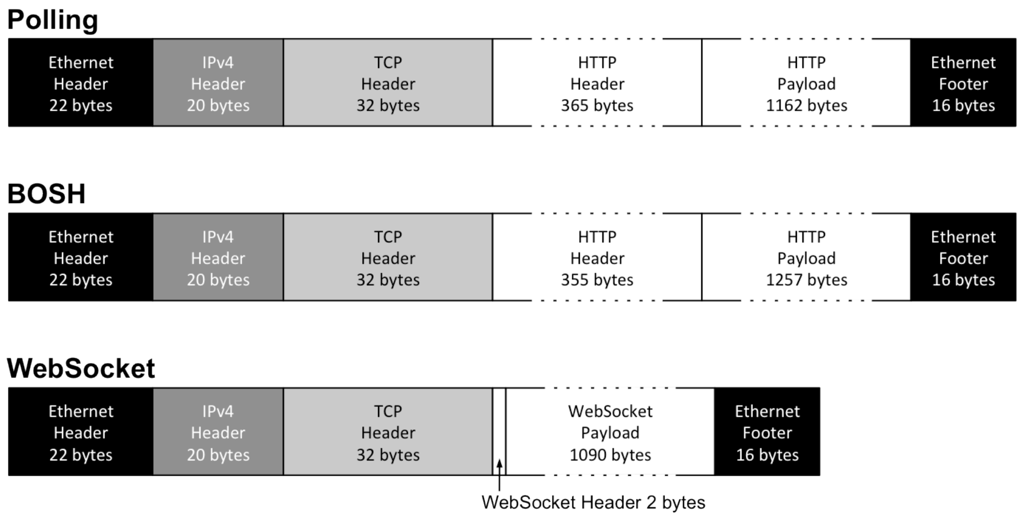

Figure 6: Ethernet frame sizes at 1000 bytes message payload size.

Round Trip Rate

The results of the round trip experiment clearly show the superiority of WebSocket in comparison to BOSH and Polling when examining single messages. Figure 7 shows the round trip rate of each technique with different message payload sizes. WebSocket achieves a round trip rate of several hundred round trips per second regardless of the message payload size. BOSH is only capable of 18 round trips per second and for Polling the rate is below 10 round trips per second. The huge gap between WebSocket and the other two techniques comes from the fact that the HTTP-based techniques are forced to make a new TCP connection to the server prior sending a new message, whereas WebSocket can reuse the same connection throughout the session. In case of Polling, also the polling interval used sets a boundary for the maximum round trip rate (with 100 ms interval the maximum round trip rate in theory is 10 messages per second).

What is interesting though, is that the round trip rate with Polling and BOSH remains constant with all of the message payload sizes, but WebSocket experiences a major drop when moving from 100 bytes message payload size to 1000 bytes message payload size. This phenomena is caused by the same factor that makes WebSocket superior in comparison to Polling and BOSH in the round trip test. Because WebSocket can use the same connection throughout the whole session, the biggest bottleneck is the size of messages transmitted between the client and the server. In other words, in Polling and BOSH the time to establish TCP connections determines the round trip rate in cases where the message can be sent in a single frame.

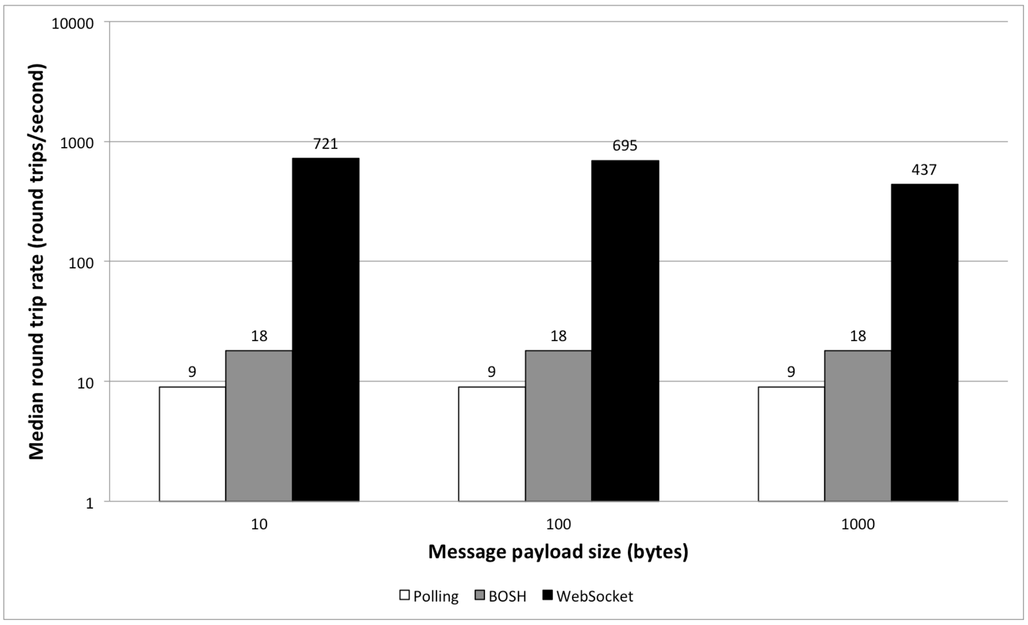

Figure 7: Round trip rate at different message payload sizes.

Message Receive Rate: Server to Client

When examining the message receive rate from the server to the client, the results seen in Figure 8 were not as dramatically in favor of WebSocket as in the round trip experiment. The WebSocket still achieved the highest message receive rate with all payloads ranging from over 2600 messages per second with 10 bytes payload to approximately 1600 messages per second with 1000 bytes payload. The range with Polling was from 2529 messages per second to 1122 messages per second, and with BOSH the range was from 1974 messages per second to 949 messages per second.

The reason for the smaller differences between the techniques in comparison to the round trip test, is that in the case of WebSocket all messages are sent as is through the TCP socket, but with Polling and BOSH the server can send multiple messages within a single response. In theory, with Polling

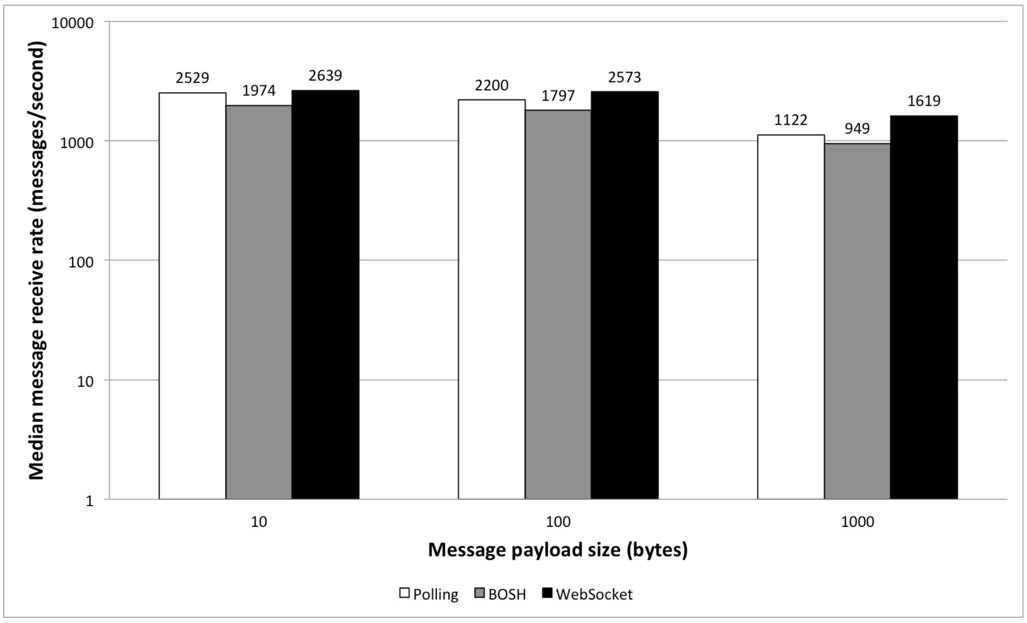

Figure 8: Server to client message rate at different message payload sizes.

and BOSH the client could receive all the 10000 messages within a single request/response pair, but in practice the amount of messages is determined by the size of the messages and the underlying network. In real-world applications, one can assume that in any network each Ethernet frame can carry up to 1500 bytes of payload, but anything bigger than 1500 bytes may or may not fit in a single frame. The maximum payload size in the network used in our setup was 15928 bytes (Jumbo Frame), meaning that in theory the Polling and BOSH clients could have received 10000 messages with 1000 bytes payload (plus XMPP overhead), making approximately 715 requests (14 XMPP messages per Jumbo Frame).

Another interesting point is that, with BOSH the message receive rate was considerably smaller in comparison to Polling with all payloads. The reason is that the BOSH client only sends a new request to the server after the server has sent a response to the previous request (i.e., new data has become available), whereas the Polling client sends requests to the server in a dense 100 ms interval in any case, which in most real-world applications would result to significant network overhead.

6 DISCUSSION

The values for message payload sizes (10, 100, and 1000 bytes) were chosen to be evenly spaced in a base-10 logarithmic scale. In addition to the aforementioned values, we planned to conduct the experiments with a message payload size of 10000 bytes. However, the message payload size of 10000 bytes was omitted from the final experiments due to unexpected technical problems, which did not come up during our pretest stage. With 10000 bytes, we could have been able to better test what would happen if the entire message does not fit into a single packet. Alternatively, more proper message payload sizes could have been gathered by observing the network traffic of typical real-time Web applications. However, we opted the former approach mainly for schedule-related reasons.

7 CONCLUSIONS AND FUTURE WORK

Real-time communication is an essential feature in a wide variety of applications. Typically, Web-based applications simulate it using HTTP-based techniques, such as polling and long-lived connections (Comet). The disadvantage of these techniques is that they are either highly inefficient, or error-prone and complex to implement. In addition, HTTP lacks uni-/multicasting support, which is a common feature requirement in related application scenarios. In this paper, we addressed these limitations by introducing how XMPP can be used on the Web. We evaluated the performance of three different XMPP communication techniques in a LAN: (1) HTTP Polling, (2) BOSH, and (3) XMPP sub-protocol for WebSocket. Our results suggest that in terms of round trip rates, XMPP sub-protocol for WebSocket performed substantially better than the HTTP Polling and BOSH techniques. In terms of message receive rates, the differences were considerably smaller in favor of XMPP sub-protocol for WebSocket. This was mainly because in the HTTP Polling and BOSH techniques multiple XMPP messages were sent within a single response. For the same reason, the actual network overhead for the HTTP Polling and BOSH techniques was much smaller than the theoretical network overhead. Otherwise, network overhead was smallest in the XMPP sub-protocol for WebSocket technique. Based on our results, we expect XMPP sub-protocol for WebSocket to gain more attention in the near future and becoming a viable option for implementing real-time communications in Web applications.

In future work, we plan to adjust and expand the experiments to cover different network environments, including both various wired and wireless network environments. Furthermore, we will conduct the experiments over both secure and non-secure connections, as recently the need for secure connections has become increasingly important. We also plan to evaluate the feasibility of the three techniques in different real-world settings with real-world Web applications in order to determine how well they deal with issues in everyday use, such as limited bandwidth and latency.

References

[1] A. Taivalsaari and T. Mikkonen. The Web as an Application Platform: The Saga Continues. In *Proceedings of the 37th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA'11)*, pages 170–174. IEEE, 2011. doi: 10.1109/SEAA.2011.35.

[2] T. Mikkonen and A. Taivalsaari. Reports of the Web's Death Are Greatly Exaggerated. *Journal of Computer*, 44(5):30–36, 2011. doi: 10.1109/MC.2011.127.

[3] P. Saint-Andre, K. Smith, and R. Tronçon. *XMPP: The Definitive Guide.* O'Reilly Media, Inc., Sebastobol, CA, the United States of America, 2009. ISBN 978-0596521264.

[4] J. Garrett. Ajax: A New Approach to Web Applications. Online. URL `http://www.adaptivepath.com/ideas/ajax-new-approach-web-applications`.

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

[6] I. Fette and A. Melnikov. The WebSocket Protocol. RFC 6455 (Proposed Standard), December 2011. URL `http://www.ietf.org/rfc/rfc6455.txt`.

[7] M. Pohja. Server Push for Web Applications via Instant Messaging. *Journal of Web Engineering*, 9(3):227–242, 2010.

[8] N. Jansen. Design and Implementation of a Web Gateway for Mobile Collaboration Services. Master's thesis, Technische Universität Dresden, 2011.

[9] K. Griffin and C. Flanagan. Evaluation of Asynchronous Event Mechanisms for Browser-based Real-time Communication Integration. In *Proceedings of Technological Developments in Networking, Education and Automation*, pages 461–466, 2010. doi: 10.1007/978-90-481-9151-2_80.

[10] E. Bozdag, A. Mesbah, and A. van Deursen. Performance Testing of Data Delivery Techniques for AJAX Applications. *Journal of Web Engineering*, 8(4):287–315, 2009.

[11] E. Bozdag, A. Mesbah, and A. van Deursen. A Comparison of Push and Pull Techniques for Ajax. In *Proceedings of the 9th IEEE International Workshop on Web Site Evolution (WSE'07)*, pages 15–22, 2007.

[12] C. Gutwin, M. Lippold, and T. Graham. Real-Time Groupware in the Browser: Testing the Performance of Web-Based Networking. In *Proceedings of the ACM 2011 conference on Computer supported cooperative work CSCW 11*, pages 167–176, 2011. doi: 10.1145/1958824.1958850.

[13] P Saint-Andre. Streaming XML with Jabber/XMPP. *IEEE Internet Computing*, 9(5):82–89, 2005. URL `http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1510608`.

[14] P. Saint-Andre. Extensible Messaging and Presence Protocol (XMPP): Core. RFC 6120 (Proposed Standard), March 2011. URL `http://www.ietf.org/rfc/rfc6120.txt`.

[15] P. Saint-Andre. Extensible Messaging and Presence Protocol (XMPP): Instant Messaging and Presence. RFC 6121 (Proposed Standard), March 2011. URL `http://www.ietf.org/rfc/rfc6121.txt`.

[16] I. Paterson and P. Saint-Andre. XEP-0206: XMPP over BOSH. Draft Standard, July 2010. URL `http://xmpp.org/extensions/xep-0206.html`.

[17] P. Millard, P. Saint-Andre, and R. Meijer. XEP-0060: Publish-Subscribe. Draft Standard, July 2010. URL `http://xmpp.org/extensions/xep-0060.html`.

[18] P. Saint-Andre. XEP-0045: Multi-User Chat. Draft Standard, February 2012. URL `http://xmpp.org/extensions/xep-0045.html`.

[19] J. Postel. Transmission Control Protocol. RFC 793 (Standard), September 1981. URL `http://www.ietf.org/rfc/rfc793.txt`. Updated by RFCs 1122, 3168, 6093, 6528.

[20] J. Hildebrand, C. Kaes, and Waite D. XEP-0025: Jabber HTTP polling. Draft Standard, June 2009. URL `http://xmpp.org/extensions/xep-0025.html`.

[21] J. Moffit and E. Cestari. An XMPP sub-protocol for WebSocket. Internet Draft, June 2011. URL `http://tools.ietf.org/html/draft-moffitt-xmpp-over-websocket-00`.