Internet Engineering Task Force (IETF)
Request for Comments: 6202
Category: Informational
ISSN: 2070-1721

S. Loreto
Ericsson
P. Saint-Andre
Cisco
S. Salsano
University of Rome "Tor Vergata"
G. Wilkins
Webtide

April 2011

Known Issues and Best Practices for the Use of Long Polling and Streaming in Bidirectional HTTP

Abstract

On today’s Internet, the Hypertext Transfer Protocol (HTTP) is often used (some would say abused) to enable asynchronous, "server-initiated" communication from a server to a client as well as communication from a client to a server. This document describes known issues and best practices related to such "bidirectional HTTP" applications, focusing on the two most common mechanisms: HTTP long polling and HTTP streaming.

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

The Hypertext Transfer Protocol [RFC2616] is a request/response protocol. HTTP defines the following entities: clients, proxies, and servers. A client establishes connections to a server for the purpose of sending HTTP requests. A server accepts connections from clients in order to service HTTP requests by sending back responses. Proxies are intermediate entities that can be involved in the delivery of requests and responses from the client to the server and vice versa.

In the standard HTTP model, a server cannot initiate a connection with a client nor send an unrequested HTTP response to a client; thus, the server cannot push asynchronous events to clients. Therefore, in order to receive asynchronous events as soon as possible, the client needs to poll the server periodically for new content. However, continual polling can consume significant bandwidth by forcing a request/response round trip when no data is available. It can also be inefficient because it reduces the responsiveness of the application since data is queued until the server receives the next poll request from the client.

In order to improve this situation, several server-push programming mechanisms have been implemented in recent years. These mechanisms, which are often grouped under the common label "Comet" [COMET], enable a web server to send updates to clients without waiting for a poll request from the client. Such mechanisms can deliver updates to clients in a more timely manner while avoiding the latency experienced by client applications due to the frequent opening and closing of connections necessary to periodically poll for data.

The two most common server-push mechanisms are HTTP long polling and HTTP streaming:

HTTP Long Polling: The server attempts to "hold open" (not immediately reply to) each HTTP request, responding only when there are events to deliver. In this way, there is always a pending request to which the server can reply for the purpose of delivering events as they occur, thereby minimizing the latency in message delivery.

HTTP Streaming: The server keeps a request open indefinitely; that is, it never terminates the request or closes the connection, even after it pushes data to the client.

It is possible to define other technologies for bidirectional HTTP; however, such technologies typically require changes to HTTP itself (e.g., by defining new HTTP methods). This document focuses only on bidirectional HTTP technologies that work within the current scope of HTTP as defined in [RFC2616] (HTTP 1.1) and [RFC1945] (HTTP 1.0).

The authors acknowledge that both the HTTP long polling and HTTP streaming mechanisms stretch the original semantic of HTTP and that the HTTP protocol was not designed for bidirectional communication. This document neither encourages nor discourages the use of these mechanisms, and takes no position on whether they provide appropriate solutions to the problem of providing bidirectional communication between clients and servers. Instead, this document merely identifies technical issues with these mechanisms and suggests best practices for their deployment.

The remainder of this document is organized as follows. Section 2 analyzes the HTTP long polling technique. Section 3 analyzes the HTTP streaming technique. Section 4 provides an overview of the specific technologies that use the server-push technique. Section 5 lists best practices for bidirectional HTTP using existing technologies.

2. HTTP Long Polling

2.1. Definition

With the traditional or "short polling" technique, a client sends regular requests to the server and each request attempts to "pull" any available events or data. If there are no events or data available, the server returns an empty response and the client waits for some time before sending another poll request. The polling frequency depends on the latency that the client can tolerate in retrieving updated information from the server. This mechanism has the drawback that the consumed resources (server processing and network) strongly depend on the acceptable latency in the delivery of updates from server to client. If the acceptable latency is low (e.g., on the order of seconds), then the polling frequency can cause an unacceptable burden on the server, the network, or both.

In contrast with such "short polling", "long polling" attempts to minimize both the latency in server-client message delivery and the use of processing/network resources. The server achieves these efficiencies by responding to a request only when a particular event, status, or timeout has occurred. Once the server sends a long poll response, typically the client immediately sends a new long poll request. Effectively, this means that at any given time the server will be holding open a long poll request, to which it replies when new information is available for the client. As a result, the server is able to asynchronously "initiate" communication.

The basic life cycle of an application using HTTP long polling is as follows:

1. The client makes an initial request and then waits for a response.

2. The server defers its response until an update is available or until a particular status or timeout has occurred.

3. When an update is available, the server sends a complete response to the client.

4. The client typically sends a new long poll request, either immediately upon receiving a response or after a pause to allow an acceptable latency period.

The HTTP long polling mechanism can be applied to either persistent or non-persistent HTTP connections. The use of persistent HTTP connections will avoid the additional overhead of establishing a new TCP/IP connection [TCP] for every long poll request.

2.2. HTTP Long Polling Issues

The HTTP long polling mechanism introduces the following issues.

Header Overhead: With the HTTP long polling technique, every long poll request and long poll response is a complete HTTP message and thus contains a full set of HTTP headers in the message framing. For small, infrequent messages, the headers can represent a large percentage of the data transmitted. If the network MTU (Maximum Transmission Unit) allows all the information (including the HTTP header) to fit within a single IP packet, this typically does not represent a significant increase in the burden for networking entities. On the other hand, the amount of transferred data can be significantly larger than the real payload carried by HTTP, and this can have a significant impact (e.g., when volume-based charging is in place).

Maximal Latency: After a long poll response is sent to a client, the server needs to wait for the next long poll request before another message can be sent to the client. This means that while the average latency of long polling is close to one network transit, the maximal latency is over three network transits (long poll response, next long poll request, long poll response). However, because HTTP is carried over TCP/IP, packet loss and retransmission can occur; therefore, maximal latency for any TCP/IP protocol will be more than three network transits (lost packet, next packet, negative ack, retransmit). When HTTP pipelining (see Section 5.2) is available, the latency due to the server waiting for a new request can be avoided.

Connection Establishment: A common criticism of both short polling and long polling is that these mechanisms frequently open TCP/IP connections and then close them. However, both polling mechanisms work well with persistent HTTP connections that can be reused for many poll requests. Specifically, the short duration of the pause between a long poll response and the next long poll request avoids the closing of idle connections.

Allocated Resources: Operating systems will allocate resources to TCP/IP connections and to HTTP requests outstanding on those connections. The HTTP long polling mechanism requires that for each client both a TCP/IP connection and an HTTP request are held open. Thus, it is important to consider the resources related to both of these when sizing an HTTP long polling application. Typically, the resources used per TCP/IP connection are minimal and can scale reasonably. Frequently, the resources allocated to HTTP requests can be significant, and scaling the total number of requests outstanding can be limited on some gateways, proxies, and servers.

Graceful Degradation: A long polling client or server that is under load has a natural tendency to gracefully degrade in performance at a cost of message latency. If load causes either a client or server to run slowly, then events to be pushed to the client will queue (waiting either for the client to send a long poll request or for the server to free up CPU cycles that can be used to process a long poll request that is being held at the server). If multiple messages are queued for a client, they might be delivered in a batch within a single long poll response. This can significantly reduce the per-message overhead and thus ease the workload of the client or server for the given message load.

Timeouts: Long poll requests need to remain pending or "hanging" until the server has something to send to the client. The timeout issues related to these pending requests are discussed in Section 5.5.

Caching: Caching mechanisms implemented by intermediate entities can interfere with long poll requests. This issue is discussed in Section 5.6.

3. HTTP Streaming

3.1. Definition

The HTTP streaming mechanism keeps a request open indefinitely. It never terminates the request or closes the connection, even after the server pushes data to the client. This mechanism significantly reduces the network latency because the client and the server do not need to open and close the connection.

The basic life cycle of an application using HTTP streaming is as follows:

1. The client makes an initial request and then waits for a response.

2. The server defers the response to a poll request until an update is available, or until a particular status or timeout has occurred.

3. Whenever an update is available, the server sends it back to the client as a part of the response.

4. The data sent by the server does not terminate the request or the connection. The server returns to step 3.

The HTTP streaming mechanism is based on the capability of the server to send several pieces of information in the same response, without terminating the request or the connection. This result can be achieved by both HTTP/1.1 and HTTP/1.0 servers.

An HTTP response content length can be defined using three options:

Content-Length header: This indicates the size of the entity body in the message, in bytes.

Transfer-Encoding header: The 'chunked' valued in this header indicates the message will break into chunks of known size if needed.

End of File (EOF): This is actually the default approach for HTTP/1.0 where the connections are not persistent. Clients do not need to know the size of the body they are reading; instead they expect to read the body until the server closes the connection. Although with HTTP/1.1 the default is for persistent connections, it is still possible to use EOF by setting the 'Connection:close' header in either the request or the response, thereby indicating that the connection is not to be considered 'persistent' after the current request/response is complete. The client's inclusion of the 'Connection: close' header field in the request will also prevent pipelining.

The main issue with EOF is that it is difficult to tell the difference between a connection terminated by a fault and one that is correctly terminated.

An HTTP/1.0 server can use only EOF as a streaming mechanism. In contrast, both EOF and "chunked transfer" are available to an HTTP/1.1 server.

The "chunked transfer" mechanism is the one typically used by HTTP/1.1 servers for streaming. This is accomplished by including the header "Transfer-Encoding: chunked" at the beginning of the response, which enables the server to send the following parts of the response in different "chunks" over the same connection. Each chunk starts with the hexadecimal expression of the length of its data, followed by CR/LF (the end of the response is indicated with a chunk of size 0).

```
HTTP/1.1 200 OK
Content-Type: text/plain
Transfer-Encoding: chunked

25
This is the data in the first chunk

1C
and this is the second one

0
```

Figure 1: Transfer-Encoding response

To achieve the same result, an HTTP/1.0 server will omit the Content-Length header in the response. Thus, it will be able to send the subsequent parts of the response on the same connection (in this case, the different parts of the response are not explicitly separated by HTTP protocol, and the end of the response is achieved by closing the connection).

3.2. HTTP Streaming Issues

The HTTP streaming mechanism introduces the following issues.

Network Intermediaries: The HTTP protocol allows for intermediaries (proxies, transparent proxies, gateways, etc.) to be involved in the transmission of a response from the server to the client. There is no requirement for an intermediary to immediately forward a partial response, and it is legal for the intermediary to buffer the entire response before sending any data to the client (e.g., caching transparent proxies). HTTP streaming will not work with such intermediaries.

Maximal Latency: Theoretically, on a perfect network, an HTTP streaming protocol’s average and maximal latency is one network transit. However, in practice, the maximal latency is higher due to network and browser limitations. The browser techniques used to terminate HTTP streaming connections are often associated with JavaScript and/or DOM (Document Object Model) elements that will grow in size for every message received. Thus, in order to avoid unlimited growth of memory usage in the client, an HTTP streaming implementation occasionally needs to terminate the streaming response and send a request to initiate a new streaming response (which is essentially equivalent to a long poll). Thus, the maximal latency is at least three network transits. Also, because HTTP is carried over TCP/IP, packet loss and retransmission can occur; therefore maximal latency for any TCP/IP protocol will be more than three network transits (lost packet, next packet, negative ack, retransmit).

Client Buffering: There is no requirement in existing HTTP specifications for a client library to make the data from a partial HTTP response available to the client application. For example, if each response chunk contains a statement of JavaScript, there is no requirement in the browser to execute that JavaScript before the entire response is received. However, in practice, most browsers do execute JavaScript received in partial responses -- although some require a buffer overflow to trigger execution. In most implementations, blocks of white space can be sent to achieve buffer overflow.

Framing Techniques: Using HTTP streaming, several application messages can be sent within a single HTTP response. The separation of the response stream into application messages needs to be performed at the application level and not at the HTTP level. In particular, it is not possible to use the HTTP chunks as application message delimiters, since intermediate proxies might "re-chunk" the message stream (for example, by combining different chunks into a longer one). This issue does not affect the HTTP long polling technique, which provides a canonical framing technique: each application message can be sent in a different HTTP response.

4. Overview of Technologies

This section provides an overview of existing technologies that implement HTTP-based server-push mechanisms to asynchronously deliver messages from the server to the client.

4.1. Bayeux

The Bayeux protocol [BAYEUX] was developed in 2006-2007 by the Dojo Foundation. Bayeux can use both the HTTP long polling and HTTP streaming mechanisms.

In order to achieve bidirectional communications, a Bayeux client will use two HTTP connections to a Bayeux server so that both server-to-client and client-to-server messaging can occur asynchronously.

The Bayeux specification requires that implementations control pipelining of HTTP requests, so that requests are not pipelined inappropriately (e.g., a client-to-server message pipelined behind a long poll request).

In practice, for JavaScript clients, such control over pipelining is not possible in current browsers. Therefore, JavaScript implementations of Bayeux attempt to meet this requirement by limiting themselves to a maximum of two outstanding HTTP requests at any one time, so that browser connection limits will not be applied and the requests will not be queued or pipelined. While broadly effective, this mechanism can be disrupted if non-Bayeux JavaScript clients simultaneously issue requests to the same host.

Bayeux connections are negotiated between client and server with handshake messages that allow the connection type, authentication method, and other parameters to be agreed upon between the client and the server. Furthermore, during the handshake phase, the client and the server reveal to each other their acceptable bidirectional techniques, and the client selects one from the intersection of those sets.

For non-browser or same-domain Bayeux, clients use HTTP POST requests to the server for both the long poll request and the request to send messages to the server. The Bayeux protocol packets are sent as the body of the HTTP messages using the "application/json" Internet media type [RFC4627].

For browsers that are operating in cross-domain mode, Bayeux attempts to use Cross-Origin Resource Sharing [CORS] checking if the browser and server support it, so that normal HTTP POST requests can be used. If this mechanism fails, Bayeux clients use the "JSONP" mechanism as described in [JSONP]. In this last case, client-to-server messages are sent as encoded JSON on the URL query parameters, and server-to-client messages are sent as a JavaScript program that wraps the message JSON with a JavaScript function call to the already loaded Bayeux implementation.

4.2. BOSH

BOSH, which stands for Bidirectional-streams Over Synchronous HTTP [BOSH], was developed by the XMPP Standards Foundation in 2003-2004. The purpose of BOSH is to emulate normal TCP connections over HTTP (TCP is the standard connection mechanism used in the Extensible Messaging and Presence Protocol as described in [RFC6120]). BOSH employs the HTTP long polling mechanism by allowing the server (called a "BOSH connection manager") to defer its response to a request until it actually has data to send to the client from the application server itself (typically an XMPP server). As soon as the client receives a response from the connection manager, it sends another request to the connection manager, thereby ensuring that the connection manager is (almost) always holding a request that it can use to "push" data to the client.

In some situations, the client needs to send data to the server while it is waiting for data to be pushed from the connection manager. To prevent data from being pipelined behind the long poll request that is on hold, the client can send its outbound data in a second HTTP request over a second TCP connection. BOSH forces the server to respond to the request it has been holding on the first connection as soon as it receives a new request from the client, even if it has no data to send to the client. It does so to make sure that the client can send more data immediately, if necessary -- even in the case where the client is not able to pipeline the requests -- while simultaneously respecting the two-connection limit discussed in Section 5.1.

The number of long poll request-response pairs is negotiated during the first request sent from the client to the connection manager. Typically, BOSH clients and connection managers will negotiate the use of two pairs, although it is possible to use only one pair or more than two pairs.

The roles of the two request-response pairs typically switch whenever the client sends data to the connection manager. This means that when the client issues a new request, the connection manager immediately answers the blocked request on the other TCP connection, thus freeing it; in this way, in a scenario where only the client sends data, the even requests are sent over one connection, and the odd ones are sent over the other connection.

BOSH is able to work reliably both when network conditions force every HTTP request to be made over a different TCP connection and when it is possible to use HTTP/1.1 and then rely on two persistent TCP connections.

If the connection manager has no data to send to the client for an agreed amount of time (also negotiated during the first request), then the connection manager will respond to the request it has been holding with no data, and that response immediately triggers a fresh client request. The connection manager does so to ensure that if a network connection is broken then both parties will realize that fact within a reasonable amount of time.

Moreover, BOSH defines the negotiation of an "inactivity period" value that specifies the longest allowable inactivity period (in seconds). This enables the client to ensure that the periods with no requests pending are never too long.

BOSH allows data to be pushed immediately when HTTP pipelining is available. However, if HTTP pipelining is not available and one of the endpoints has just pushed some data, BOSH will usually need to wait for a network round-trip time until the server is able to again push data to the client.

BOSH uses standard HTTP POST request and response bodies to encode all information.

BOSH normally uses HTTP pipelining over a persistent HTTP/1.1 connection. However, a client can deliver its POST requests in any way permitted by HTTP 1.0 or HTTP 1.1. (Although the use of HTTP POST with pipelining is discouraged in RFC 2616, BOSH employs various methods, such as request identifiers, to ensure that this usage does not lead to indeterminate results if the transport connection is terminated prematurely.)

BOSH clients and connection managers are not allowed to use Chunked Transfer Coding, since intermediaries might buffer each partial HTTP request or response and only forward the full request or response once it is available.

BOSH allows the usage of the Accept-Encoding and Content-Encoding headers in the request and in the response, respectively, and then compresses the response body accordingly.

Each BOSH session can share the HTTP connection(s) it uses with other HTTP traffic, including other BOSH sessions and HTTP requests and responses completely unrelated to the BOSH protocol (e.g., Web page downloads).

4.3. Server-Sent Events

W3C Server-Sent Events specification [WD-eventsource] defines an API that enables servers to push data to Web pages over HTTP in the form of DOM events.

The data is encoded as "text/event-stream" content and pushed using an HTTP streaming mechanism, but the specification suggests disabling HTTP chunking for serving event streams unless the rate of messages is high enough to avoid the possible negative effects of this technique as described in Section 3.2.

However, it is not clear if there are significant benefits to using EOF rather than chunking with regards to intermediaries, unless they support only HTTP/1.0.

5. HTTP Best Practices

5.1. Limits to the Maximum Number of Connections

HTTP [RFC2616], Section 8.1.4, recommends that a single user client not maintain more than two connections to any server or proxy, in order to prevent the server from being overloaded and to avoid unexpected side effects in congested networks. Until recently, this limit was implemented by most commonly deployed browsers, thus making connections a scarce resource that needed to be shared within the browser. Note that the available JavaScript APIs in the browsers hide the connections, and the security model inhibits the sharing of any resource between frames. The new HTTP specification [HTTPBIS] removes the two-connection limitation, only encouraging clients to be conservative when opening multiple connections. In fact, recent browsers have increased this limit to 6 or 8 connections; however, it is still not possible to discover the local limit, and usage of multiple frames and tabs still places 8 connections within easy reach.

Web applications need to limit the number of long poll requests initiated, ideally to a single long poll that is shared between frames, tabs, or windows of the same browser. However, the security constraints of the browsers make such sharing difficult.

A best practice for a server is to use cookies [COOKIE] to detect multiple long poll requests from the same browser and to avoid deferring both requests since this might cause connection starvation and/or pipeline issues.

5.2. Pipelined Connections

HTTP [RFC2616] permits optional request pipelining over persistent connections. Multiple requests can be enqueued before the responses arrive.

In the case of HTTP long polling, the use of HTTP pipelining can reduce latency when multiple messages need to be sent by a server to a client in a short period of time. With HTTP pipelining, the server can receive and enqueue a set of HTTP requests. Therefore, the server does not need to receive a new HTTP request from the client after it has sent a message to the client within an HTTP response. In principle, the HTTP pipelining can be applied to HTTP GET and HTTP POST requests, but using HTTP POST requests is more critical. In fact, the use of HTTP POST with pipelining is discouraged in RFC 2616 and needs to be handled with special care.

There is an issue regarding the inability to control pipelining. Normal requests can be pipelined behind a long poll, and are thus delayed until the long poll completes.

Mechanisms for bidirectional HTTP that want to exploit HTTP pipelining need to verify that HTTP pipelining is available (e.g., supported by the client, the intermediaries, and the server); if it’s not available, they need to fall back to solutions without HTTP pipelining.

5.3. Proxies

Most proxies work well with HTTP long polling because a complete HTTP response will be sent either on an event or a timeout. Proxies are advised to return that response immediately to the user agent, which immediately acts on it.

The HTTP streaming mechanism uses partial responses and sends some JavaScript in an HTTP/1.1 chunk as described in Section 3. This mechanism can face problems caused by two factors: (1) it relies on proxies to forward each chunk (even though there is no requirement for them to do so, and some caching proxies do not), and (2) it relies on user agents to execute the chunk of JavaScript as it arrives (even though there is also no requirement for them to do so).

A "reverse proxy" basically is a proxy that pretends to be the actual server (as far as any client or client proxy is concerned), but it passes on the request to the actual server that is usually sitting behind another layer of firewalls. Any HTTP short polling or HTTP long polling solution will work fine with this, as will most HTTP streaming solutions. The main downside is performance, since most proxies are not designed to hold many open connections.

Reverse proxies can come to grief when they try to share connections to the servers between multiple clients. As an example, Apache with mod_jk shares a small set of connections (often 8 or 16) between all clients. If long polls are sent on those shared connections, then the proxy can be starved of connections, which means that other requests (either long poll or normal) can be held up. Thus, Comet mechanisms currently need to avoid any connection sharing -- either in the browser or in any intermediary -- because the HTTP assumption is that each request will complete as fast as possible.

One of the main reasons why both HTTP long polling and HTTP streaming are perceived as having a negative impact on servers and proxies is that they use a synchronous programming model for handling requests, since the resources allocated to each request are held for the duration of the request. Asynchronous proxies and servers can handle long polls using slightly more resources than normal HTTP traffic. Unfortunately some synchronous proxies do exist (e.g., Apache mod_jk) and many HTTP application servers also have a blocking model for their request handling (e.g., the Java servlet 2.5 specification).

5.4. HTTP Responses

In accordance with [RFC2616], the server responds to a request it has successfully received by sending a 200 OK answer, but only when a particular event, status, or timeout has occurred. The 200 OK body section contains the actual event, status, or timeout that occurred. This "best practice" is simply standard HTTP.

5.5. Timeouts

The HTTP long polling mechanism allows the server to respond to a request only when a particular event, status, or timeout has occurred. In order to minimize (as much as possible) both latency in server-client message delivery and the processing/network resources needed, the long poll request timeout ought to be set to a high value.

However, the timeout value has to be chosen carefully; indeed, problems can occur if this value is set too high (e.g., the client might receive a 408 Request Timeout answer from the server or a 504 Gateway Timeout answer from a proxy). The default timeout value in a browser is 300 seconds, but most network infrastructures include proxies and servers whose timeouts are not that long.

Several experiments have shown success with timeouts as high as 120 seconds, but generally 30 seconds is a safer value. Therefore, vendors of network equipment wishing to be compatible with the HTTP long polling mechanism are advised to implement a timeout substantially greater than 30 seconds (where "substantially" means several times more than the medium network transit time).

5.6. Impact on Intermediary Entities

There is no way for an end client or host to signal to HTTP intermediaries that long polling is in use; therefore, long poll requests are completely transparent for intermediary entities and are handled as normal requests. This can have an impact on intermediary entities that perform operations that are not useful in case of long polling. However, any capabilities that might interfere with bidirectional flow (e.g., caching) can be controlled with standard headers or cookies.

As a best practice, caching is always intentionally suppressed in a long poll request or response, i.e., the "Cache-Control" header is set to "no-cache".

6. Security Considerations

This document is meant to describe current usage of HTTP to enable asynchronous or server-initiated communication. It does not propose any change to the HTTP protocol or to the expected behavior of HTTP entities. Therefore this document does not introduce new security concerns into existing HTTP infrastructure. The considerations reported hereafter refer to the solutions that are already implemented and deployed.

One security concern with cross-domain HTTP long polling is related to the fact that often the mechanism is implemented by executing the JavaScript returned from the long poll request. If the server is prone to injection attacks, then it could be far easier to trick a browser into executing the code [CORS].

Another security concern is that the number of open connections that needs to be maintained by a server in HTTP long polling and HTTP streaming could more easily lead to denial-of-service (DoS) attacks [RFC4732].

7. References

7.1. Normative References

[RFC1945] Berners-Lee, T., Fielding, R., and H. Nielsen, "Hypertext Transfer Protocol -- HTTP/1.0", RFC 1945, May 1996.

[RFC2616] Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., and T. Berners-Lee, "Hypertext Transfer Protocol -- HTTP/1.1", RFC 2616, June 1999.

[RFC4732] Handley, M., Rescorla, E., and IAB, "Internet Denial-of-Service Considerations", RFC 4732, December 2006.

7.2. Informative References

[BAYEUX] Russell, A., Wilkins, G., Davis, D., and M. Nesbitt, "Bayeux Protocol -- Bayeux 1.0.0", 2007, <http://svn.cometd.com/trunk/bayeux/bayeux.html>.

[BOSH] Paterson, I., Smith, D., and P. Saint-Andre, "Bidirectional-streams Over Synchronous HTTP (BOSH)", XSF XEP 0124, February 2007.

[COMET] Russell, A., "Comet: Low Latency Data for the Browser", March 2006, <http://infrequently.org/2006/03/comet-low-latency-data-for-the-browser/ >.

[COOKIE] Barth, A., "HTTP State Management Mechanism", Work in Progress, March 2011.

[CORS] van Kesteren, A., "Cross-Origin Resource Sharing", W3C Working Draft WD-cors-20100727, latest version available at <http://www.w3.org/TR/cors/>, July 2010, <http://www.w3.org/TR/2010/WD-cors-20100727/>.

[HTTPBIS] Fielding, R., Ed., Gettys, J., Mogul, J., Nielsen, H., Masinter, L., Leach, P., Berners-Lee, T., Lafon, Y., Ed., and J. Reschke, Ed., "HTTP/1.1, part 1: URIs, Connections, and Message Parsing", Work in Progress, March 2011.

[JSONP] Wikipedia, "JSON with padding", <http://en.wikipedia.org/wiki/JSONP#JSONP>.

[RFC4627] Crockford, D., "The application/json Media Type for JavaScript Object Notation (JSON)", RFC 4627, July 2006.

[RFC6120] Saint-Andre, P., "Extensible Messaging and Presence Protocol (XMPP): Core", RFC 6120, March 2011.

[TCP] Postel, J., "Transmission Control Protocol", STD 7, RFC 793, September 1981.

[WD-eventsource] Hickson, I., "Server-Sent Events", W3C Working Draft WD-eventsource-20091222, latest version available at <http://www.w3.org/TR/eventsource/>, December 2009, <http://www.w3.org/TR/2009/WD-eventsource-20091222/>.

8. Acknowledgments

Thanks to Joe Hildebrand, Julien Laganier, Jack Moffitt, Subramanian Moonesamy, Mark Nottingham, Julian Reschke, Martin Thomson, and Martin Tyler for their feedback.

Authors' Addresses

Salvatore Loreto
Ericsson
Hirsalantie 11
Jorvas 02420
Finland

EMail: salvatore.loreto@ericsson.com

Peter Saint-Andre
Cisco
1899 Wyknoop Street, Suite 600
Denver, CO 80202
USA

Phone: +1-303-308-3282
EMail: psaintan@cisco.com

Stefano Salsano
University of Rome "Tor Vergata"
Via del Politecnico, 1
Rome 00133
Italy

EMail: stefano.salsano@uniroma2.it

Greg Wilkins
Webtide

EMail: gregw@webtide.com