HyBi Working Group
Internet-Draft
Intended status: Standards Track
Expires: May 13, 2011

A. Barth
Google, Inc.

November 9, 2010

The WebSocket protocol
draft-abarth-websocket-handshake-01

Abstract

The WebSocket protocol enables two-way communication between a user agent running untrusted code running in a controlled environment to a remote host that has opted-in to communications from that code. The security model used for this is the Origin-based security model commonly used by Web browsers. The protocol consists of an initial handshake followed by basic message framing, layered over TCP. The goal of this technology is to provide a mechanism for browser-based applications that need two-way communication with servers that does not rely on opening multiple HTTP connections (e.g. using XMLHttpRequest or <iframe>s and long polling).

Please send feedback to the hybi@ietf.org mailing list.

Status of this Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at http://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on May 13, 2011.

Table of Contents

1. Opening Handshake

1.1. Client Requirements

User agents running in controlled environments, e.g. browsers on mobile handsets tied to specific carriers, may offload the management of the connection to another agent on the network. In such a situation, the user agent for the purposes of conformance is considered to include both the handset software and any such agents.

When the user agent is to *establish a WebSocket connection* to a host /host/, on a port /port/, from an origin whose ASCII serialization is /origin/, with a flag /secure/, with a string giving a /resource name/, with a (possibly empty) list of strings giving the /protocols/, and optionally with a /defer cookies/ flag, it must run the following steps. [ORIGIN]

1. Verify that the WebSocket URL and its components are valid according to ???. If any of the requirements are not met, the client MUST fail the WebSocket connection and abort these steps.

2. If the user agent already has a WebSocket connection to the remote host (IP address) identified by /host/, even if known by another name, wait until that connection has been established or for that connection to have failed. If multiple connections to the same IP address are attempted simultaneously, the user agent must serialize them so that there is no more than one connection at a time running through the following steps.

   If the user agent cannot determine the IP address of the remote host (for example because all communication is being done through a proxy server that performs DNS queries itself), then the user agent must assume for the purposes of this step that each host name refers to a distinct remote host, but should instead limit the total number of simultaneous connections that are not established to a reasonably low number (e.g., in a Web browser, to the number of tabs the user has open).

   NOTE: This makes it harder for a script to perform a denial of service attack by just opening a large number of WebSocket connections to a remote host. A server can further reduce the load on itself when attacked by making use of this by pausing before closing the connection, as that will reduce the rate at which the client reconnects.

   NOTE: There is no limit to the number of established WebSocket connections a user agent can have with a single remote host. Servers can refuse to connect users with an excessive number of

connections, or disconnect resource-hogging users when suffering high load.

3. _Connect_: If the user agent is configured to use a proxy when using the WebSocket protocol to connect to host /host/ and/or port /port/, then connect to that proxy and ask it to open a TCP connection to the host given by /host/ and the port given by /port/.

   EXAMPLE: For example, if the user agent uses an HTTP proxy for all traffic, then if it was to try to connect to port 80 on server example.com, it might send the following lines to the proxy server:

```
CONNECT example.com:80 HTTP/1.1
Host: example.com
```

   If there was a password, the connection might look like:

```
CONNECT example.com:80 HTTP/1.1
Host: example.com
Proxy-authorization: Basic ZWRuYW1vZGU6bm9jYXBlcyE=
```

   Otherwise, if the user agent is not configured to use a proxy, then open a TCP connection to the host given by /host/ and the port given by /port/.

   NOTE: Implementations that do not expose explicit UI for selecting a proxy for WebSocket connections separate from other proxies are encouraged to use a SOCKS proxy for WebSocket connections, if available, or failing that, to prefer the proxy configured for HTTPS connections over the proxy configured for HTTP connections.

   For the purpose of proxy autoconfiguration scripts, the URL to pass the function must be constructed from /host/, /port/, /resource name/, and the /secure/ flag using the steps to construct a WebSocket URL.

   NOTE: The WebSocket protocol can be identified in proxy autoconfiguration scripts from the scheme ("ws:" for unencrypted connections and "wss:" for encrypted connections).

4. If the connection could not be opened, then fail the WebSocket connection and abort these steps.

5. If /secure/ is true, perform a TLS handshake over the connection. If this fails (e.g. the server's certificate could not be verified), then fail the WebSocket connection and abort these steps. Otherwise, all further communication on this channel must run through the encrypted tunnel. [RFC2246]

   User agents must use the Server Name Indication extension in the TLS handshake. [RFC4366]

6. Let the client-nonce be a 16 byte sequence chosen uniformly at random.

7. Send the following strings (in order):

   1. Send the UTF-8 string "CONNECT websocket.invalid:443 HTTP/1.1".

   2. Send a UTF-8-encoded U+000D CARRIAGE RETURN U+000A LINE FEED character pair (CRLF).

   3. Send the UTF-8 string "Host: websocket.invalid:443".

   4. Send a UTF-8-encoded U+000D CARRIAGE RETURN U+000A LINE FEED character pair (CRLF).

   5. Send the UTF-8 string "Sec-WebSocket-Key: ".

   6. Send the client-nonce encoded in base64.

   7. Send a UTF-8-encoded U+000D CARRIAGE RETURN U+000A LINE FEED character pair (CRLF).

   8. Send the UTF-8 string "Sec-WebSocket-Metadata: ".

8. Let /hostport/ be an empty string.

9. Append the /host/ value, converted to ASCII lowercase, to /hostport/.

10. If /secure/ is false, and /port/ is not 80, or if /secure/ is true, and /port/ is not 443, then append a U+003A COLON character (:) followed by the value of /port/, expressed as a base-ten integer, to /hostport/.

11. Let the metadata-dictionary be the following dictionary:

    1. Key "resource_name" maps to value /resource name/.

2. Key "host" maps to value /host/.

3. Key "origin" maps to value /origin/.

4. Key "protocols" maps to the array containing each protocol in /protocols/.

5. If the client has any cookies that would be relevant to a resource accessed over HTTP, if /secure/ is false, or HTTPS, if it is true, on host /host/, port /port/, with /resource name/ as the path (and possibly query parameters), then key "cookie" maps to the cookie-string for that resource (including http-only cookies).[RFC2616][RFC2109][RFC2965]

12. Let the metadata-string be the JSON serialization of the metadata-dictionary in UTF-8.

13. Let the handshake-mask be the HMAC-SHA1 of the UTF-8 string "C1BA787A-0556-49F3-B6AE-32E5376F992B" keyed with the client-nonce.

14. Let the masked-metadata be a sequence of bytes where the /i/th byte is the XOR of the /i/th byte of the metadata-string with the /i mod 20/th byte of the handshake-mask.

15. Send the following strings (in order):

    1. Send the masked metadata encoded in base64.

    2. Send a UTF-8-encoded U+000D CARRIAGE RETURN U+000A LINE FEED character pair (CRLF).

    3. Send a UTF-8-encoded U+000D CARRIAGE RETURN U+000A LINE FEED character pair (CRLF).

16. Read bytes from the server until either the connection closes or until 17 bytes are read. If the read bytes are not exactly the UTF-8 string "HTTP/1.1 200 OK" followed by a UTF-8-encoded U+000D CARRIAGE RETURN U+000A LINE FEED character pair (CRLF), then _fail the WebSocket connection_ and abort these steps.

    User agents may apply a timeout to this step, failing the WebSocket connection if the server does not send back data in a suitable time period.

17. Let /fields/ be a list of name-value pairs, initially empty.

18. _Field_: Let /name/ and /value/ be empty byte arrays.

19. Read bytes from the server.

    If the connection closes before this byte is received, then fail the WebSocket connection and abort these steps.

    Otherwise, handle the byte as described in the appropriate entry below:

    -> If the byte is 0x0D (UTF-8 CR)
       If the /name/ byte array is empty, then jump to the fields processing step. Otherwise, fail the WebSocket connection and abort these steps.

    -> If the byte is 0x0A (UTF-8 LF)
       Fail the WebSocket connection and abort these steps.

    -> If the byte is 0x3A (UTF-8 :)
       Move on to the next step.

    -> If the byte is in the range 0x41 to 0x5A (UTF-8 A-Z)
       Append a byte whose value is the byte's value plus 0x20 to the /name/ byte array and redo this step for the next byte.

    -> Otherwise
       Append the byte to the /name/ byte array and redo this step for the next byte.

    NOTE: This reads a field name, terminated by a colon, converting upper-case letters in the range A-Z to lowercase, and aborting if a stray CR or LF is found.

20. Let /count/ equal 0.

    NOTE: This is used in the next step to skip past a space character after the colon, if necessary.

21. Read a byte from the server and increment /count/ by 1.

    If the connection closes before this byte is received, then fail the WebSocket connection and abort these steps.

    Otherwise, handle the byte as described in the appropriate entry below:

-> If the byte is 0x20 (UTF-8 space) and /count/ equals 1
Ignore the byte and redo this step for the next byte.

-> If the byte is 0x0D (UTF-8 CR)
Move on to the next step.

-> If the byte is 0x0A (UTF-8 LF)
Fail the WebSocket connection and abort these steps.

-> Otherwise
Append the byte to the /value/ byte array and redo this step for the next byte.

NOTE: This reads a field value, terminated by a CRLF, skipping past a single space after the colon if there is one.

22. Read a byte from the server.

    If the connection closes before this byte is received, or if the byte is not a 0x0A byte (UTF-8 LF), then fail the WebSocket connection and abort these steps.

    NOTE: This skips past the LF byte of the CRLF after the field.

23. Append an entry to the /fields/ list that has the name given by the string obtained by interpreting the /name/ byte array as a UTF-8 byte stream and the value given by the string obtained by interpreting the /value/ byte array as a UTF-8 byte stream.

24. Return to the "Field" step above.

25. _Fields processing_: Read a byte from the server.

    If the connection closes before this byte is received, or if the byte is not a 0x0A byte (UTF-8 LF), then fail the WebSocket connection and abort these steps.

    NOTE: This skips past the LF byte of the CRLF after the blank line after the fields.

26. Let the /list of cookies/ be empty.

27. If there is not exactly one entry in the /fields/ list whose name is "sec-websocket-accept", or if there is not exactly one entry in the /fields/ list whose name is "sec-websocket-key", or if there is not exactly one entry in the /fields/ list whose name is "sec-websocket-origin", or if there is not exactly one entry in the /fields/ list whose name is "sec-websocket-

location", or if the /protocol/ was specified but there is not exactly one entry in the /fields/ list whose name is "sec-websocket-protocol", or if there are any entries in the /fields/ list whose names are the empty string, then fail the WebSocket connection and abort these steps. Otherwise, handle each entry in the /fields/ list as follows:

-> If the entry's name is "sec-websocket-accept"
   If the value is not exactly equal to the base64 encoding of the HMAC-SHA1 of the UTF-8 string "258EAFA5-E914-47DA-95CA-C5AB0DC85B11" keyed with the client-nonce, then _fail the WebSocket connection_ and abort these steps.

-> If the entry's name is "sec-websocket-key"
   If the read bytes are not the base64 encoding of a 16 byte sequence, then _fail the WebSocket connection_ and abort these steps. Otherwise, let the server-nonce be that 16 byte sequence. If the server-nonce is identical to the client-nonce, then _fail the WebSocket connection_ and abort these steps.

-> If the entry's name is "sec-websocket-origin"
   If the value is not exactly equal to /origin/, then fail the WebSocket connection and abort these steps. [ORIGIN]

-> If the entry's name is "sec-websocket-location"
   If the value is not exactly equal to a string obtained from the steps to construct a WebSocket URL from /host/, /port/, /resource name/, and the /secure/ flag, then fail the WebSocket connection and abort these steps.

-> If the entry's name is "sec-websocket-protocol"
   If there was a /protocols/ string specified, and the value is not exactly equal to one of the items in /protocols/, then fail the WebSocket connection and abort these steps. (If no /protocols/ was specified, the field is ignored.)

-> If the entry's name is "set-cookie" or "set-cookie2" or another cookie-related field name
   If the relevant specification is supported by the user agent, add the cookie, interpreted as defined by the appropriate specification, to the /list of cookies/, with the resource being the one with the host /host/, the port /port/, the path (and possibly query parameters) /resource name/, and the scheme |http| if /secure/ is false and |https| if /secure/ is true. [RFC2109] [RFC2965]

   If the relevant specification is not supported by the user

agent, then the field must be ignored.

The cookies added to the /list of cookies/ are discarded if the connection fails to be established. Only if and when the connection is established do the cookies actually get applied.

-> Any other name
Ignore it.

28. If the /defer cookies/ flag is not set, apply the cookies in the /list of cookies/.

29. The *WebSocket connection is established*. Now the user agent must send and receive to and from the connection as described in the next section. The handshake has established two keys:

    * The client-to-server-mask is the intial 16 bytes of the HMAC-SHA1 of the UTF-8 string "363A6078-74D2-4C0B-8CBC-1E6A36E83442" keyed with the concatenation of the client-nonce and the server-nonce.

    * The server-to-client-mask is the initial 16 bytes of the HMAC-SHA1 of the UTF-8 string "2306C3BE-0ACF-42C0-B69E-DFFE02CFA346" keyed with the concatentation of the client-nonce and the server-nonce.

    All subsequent bytes sent from the user agent to the server are masked as follows:

    3. The /i/th byte is XORed with the /i mod 20/th byte of the client-to-server-mask.

    All subsequence bytes read by the user agent from the server are unmasked as follows:

    4. The /i/th byte is XORed with the /i mod 20/th byte of the server-to-client-mask.

30. If the /defer cookies/ flag is set, store the /list of cookies/ for use by the component that invoked this algorithm.

Where the algorithm above requires that a user agent fail the WebSocket connection, the user agent may first read an arbitrary number of further bytes from the connection (and then discard them) before actually *failing the WebSocket connection*. Similarly, if a user agent can show that the bytes read from the connection so far

are such that there is no subsequent sequence of bytes that the server can send that would not result in the user agent being required to *fail the WebSocket connection*, the user agent may immediately *fail the WebSocket connection* without waiting for those bytes.

NOTE: The previous paragraph is intended to make it conforming for user agents to implement the algorithm in subtlely different ways that are equivalent in all ways except that they terminate the connection at earlier or later points. For example, it enables an implementation to buffer the entire handshake response before checking it, or to verify each field as it is received rather than collecting all the fields and then checking them as a block.

When the user agent is to "apply the cookies" in a /list of cookies/, it must handle each cookie in the /list of cookies/ as defined by the appropriate specification. [RFC2109] [RFC2965]

1.2. Server-side requirements

This section only applies to servers.

Servers may offload the management of the connection to other agents on the network, for example load balancers and reverse proxies. In such a situation, the server for the purposes of conformance is considered to include all parts of the server-side infrastructure from the first device to terminate the TCP connection all the way to the server that processes requests and sends responses.

EXAMPLE: For example, a data center might have a server that responds to Web Socket requests with an appropriate handshake, and then passes the connection to another server to actually process the data frames. For the purposes of this specification, the "server" is the combination of both computers.

1.2.1. Reading the client’s opening handshake

When a client starts a WebSocket connection, it sends its part of the opening handshake. The server must parse at least part of this handshake in order to obtain the necessary information to generate the server part of the handshake.

The client handshake consists of the following parts. If the server, while reading the handshake, finds that the client did not send a handshake that matches the description below, the server should abort the WebSocket connection.

1. The UTF-8 string "CONNECT websocket.invalid:443 HTTP/1.1".

2. A UTF-8-encoded U+000D CARRIAGE RETURN U+000A LINE FEED character pair (CRLF).

3. The UTF-8 string "Host: websocket.invalid:443".

4. A UTF-8-encoded U+000D CARRIAGE RETURN U+000A LINE FEED character pair (CRLF).

5. The UTF-8 string "Sec-WebSocket-Key: ".

6. A string of base64 encoded bytes terminated by a UTF-8-encoded U+000D CARRIAGE RETURN U+000A LINE FEED character pair (CRLF). The decoded bytes are the client-nonce.

7. The UTF-8 string "Sec-WebSocket-Metadata: ".

8. A string of base64 encoded bytes terminated by a UTF-8-encoded U+000D CARRIAGE RETURN U+000A LINE FEED character pair (CRLF). The decoded bytes are the masked-metadata.

Let the handshake-mask be the HMAC-SHA1 of the UTF-8 string "C1BA787A-0556-49F3-B6AE-32E5376F992B" keyed with the client-nonce.

Let the metadata-string be the masked-metadata unmasked by XORing the /i/th byte of the masked-metadata with the /i mod 20/th byte of the handshake-mask.

Let the metadata-dictionary by the result of parsing the metadata-string as a UTF-8 encoded JSON string.

The expected dictionary keys, and the meaning of their corresponding values, are as follows.

|host|
: The value gives the hostname that the client intended to use when opening the WebSocket. It would be of interest in particular to virtual hosting environments, where one server might serve multiple hosts, and might therefore want to return different data.

  Can be safely ignored, though the server should abort the WebSocket connection if this field is absent or has a value that does not match the server’s host name, to avoid vulnerability to cross-protocol attacks and DNS rebinding attacks.

|origin|
The value gives the scheme, hostname, and port (if it's not the default port for the given scheme) of the page that asked the client to open the WebSocket. It would be interesting if the server's operator had deals with operators of other sites, since the server could then decide how to respond (or indeed, _whether_ to respond) based on which site was requesting a connection. [ORIGIN]

Can be safely ignored, though the server should abort the WebSocket connection if this field is absent or has a value that does not match one of the origins the server is expecting to communicate with, to avoid vulnerability to cross-protocol attacks and cross-site scripting attacks.

|protocols|
The value gives an array of the subprotocols that the client is intending to select. It would be interesting if the server supports multiple protocols or protocol versions.

Can be safely ignored, though the server may abort the WebSocket connection if the field is absent but the conventions for communicating with the server are such that the field is expected; and the server should abort the WebSocket connection if the field has a value that does not match one of the subprotocols that the server supports, to avoid integrity errors once the connection is established.

Other keys
Other fields can be used, such as "cookie", for authentication purposes. Their semantics are equivalent to the semantics of the HTTP headers with the same names.

Unrecognized fields can be safely ignored, and are probably either the result of clients that support future versions of the protocol offering options that the server doesn't support.

1.2.2. Sending the server's opening handshake

When a client establishes a WebSocket connection to a server, the server must run the following steps.

1. If the server supports encryption, perform a TLS handshake over the connection. If this fails (e.g. the client indicated a host name in the extended client hello "server_name" extension that the server does not host), then close the connection; otherwise, all further communication for the connection (including the server handshake) must run through the encrypted tunnel.

[RFC2246]

2. Establish the following information:

/host/
The host name or IP address of the WebSocket server, as it is to be addressed by clients. The host name must be punycode-encoded if necessary. If the server can respond to requests to multiple hosts (e.g. in a virtual hosting environment), then the value should be derived from the client's handshake, specifically from the "Host" field. The /host/ value must be lowercase (not containing characters in the range U+0041 LATIN CAPITAL LETTER A to U+005A LATIN CAPITAL LETTER Z).

/port/
The port number on which the server expected and/or received the connection.

/resource name/
An identifier for the service provided by the server. If the server provides multiple services, then the value should be derived from the resource name given in the client's handshake.

/secure flag/

True if the connection is encrypted or if the server expected it to be encrypted; false otherwise.

/origin/
The ASCII serialization of the origin that the server is willing to communicate with, converted to ASCII lowercase. If the server can respond to requests from multiple origins (or indeed, all origins), then the value should be derived from the client's handshake, specifically from the "Origin" field. [ORIGIN]

/subprotocol/
Either null, or a string representing the subprotocol the server is ready to use. If the server supports multiple subprotocols, then the value should be derived from the client's handshake, specifically by selecting one of the values from the "protocols" array. The absence of such a field is equivalent to the null value. The empty string is not the same as the null value for these purposes.

3. Let /location/ be the string that results from constructing a WebSocket URL from /host/, /port/, /resource name/, and /secure flag/.

4. Let acceptance-proof be the base64 encoding of the HMAC-SHA1 of the UTF-8 string "258EAFA5-E914-47DA-95CA-C5AB0DC85B11" keyed with the client-nonce.

5. Let server-nonce be a sequence of 16 bytes chosen uniformly at random.

6. Send the following line, terminated by the two characters U+000D CARRIAGE RETURN and U+000A LINE FEED (CRLF) and encoded as UTF-8, to the client:

```
HTTP/1.1 200 OK
```

7. Send the following fields to the client. Each field must be sent as a line consisting of the field name, which must be an ASCII case-insensitive match for the field name in the list below, followed by a U+003A COLON character (:) and a U+0020 SPACE character, followed by the field value as specified in the list below, followed by the two characters U+000D CARRIAGE RETURN and U+000A LINE FEED (CRLF). The lines must be encoded as UTF-8. The lines may be sent in any order.

   |Sec-WebSocket-Accept|
   The value must be the acceptance-proof.

   |Sec-WebSocket-Key|
   The value must be the server-nonce encoded in base64.

   |Sec-WebSocket-Location|
   The value must be /location/

   |Sec-WebSocket-Origin|
   The value must be /origin/

   |Sec-WebSocket-Protocol|
   This field must be included if /subprotocol/ is not null, and must not be included if /subprotocol/ is null.

   If included, the value must be /subprotocol/

   Optionally, include "Set-Cookie", "Set-Cookie2", or other cookie-related fields, with values equal to the values that would be used for the identically named HTTP headers. [RFC2109] [RFC2965]

8. Send two bytes 0x0D 0x0A (UTF-8 CRLF).

9. Send /response/.

This completes the server's handshake. If the server finishes these steps without aborting the WebSocket connection, and if the client does not then fail the WebSocket connection, then the connection is established and the server may begin sending and receiving data, as described in the next section. The handshake has established two keys:

- The client-to-server-mask is the HMAC-SHA1 of the UTF-8 string "363A6078-74D2-4C0B-8CBC-1E6A36E83442" keyed with the concatenation of the client-nonce and the server-nonce.

- The server-to-client-mask is the HMAC-SHA1 of the UTF-8 string "2306C3BE-0ACF-42C0-B69E-DFFE02CFA346" keyed with the concatentation of the client-nonce and the server-nonce.

All subsequent bytes read by the server from the user agent are unmasked as follows:

> The /i/th byte is XORed with the /i mod 20/th byte of the client-to-server-mask.

All subsequence bytes sent from the server to the user agent are masked as follows:

> The /i/th byte is XORed with the /i mod 20/th byte of the server-to-client-mask.

2. Normative References

[HTML] Hickson, I., "HTML", August 2010, <http://whatwg.org/html5>.

[ORIGIN] Barth, A., Jackson, C., and I. Hickson, "The HTTP Origin Header", draft-abarth-origin (work in progress), September 2009, <http://tools.ietf.org/html/draft-abarth-origin>.

[ANSI.X3-4.1986] American National Standards Institute, "Coded Character Set - 7-bit American Standard Code for Information Interchange", ANSI X3.4, 1986.

[RFC1321] Rivest, R., "The MD5 Message-Digest Algorithm", RFC 1321, April 1992.

[RFC1951] Deutsch, P., "DEFLATE Compressed Data Format Specification version 1.3", RFC 1951, May 1996.

[RFC2109] Kristol, D. and L. Montulli, "HTTP State Management Mechanism", RFC 2109, February 1997.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

[RFC2246] Dierks, T. and C. Allen, "The TLS Protocol Version 1.0", RFC 2246, January 1999.

[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.

[RFC2965] Kristol, D. and L. Montulli, "HTTP State Management Mechanism", RFC 2965, October 2000.

[RFC3490] Faltstrom, P., Hoffman, P., and A. Costello, "Internationalizing Domain Names in Applications (IDNA)", RFC 3490, March 2003.

[RFC3629] Yergeau, F., "UTF-8, a transformation format of ISO 10646", STD 63, RFC 3629, November 2003.

[RFC3864] Klyne, G., Nottingham, M., and J. Mogul, "Registration Procedures for Message Header Fields", BCP 90, RFC 3864, September 2004.

[RFC3986] Berners-Lee, T., Fielding, R., and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax", STD 66, RFC 3986, January 2005.

[RFC3987] Duerst, M. and M. Suignard, "Internationalized Resource Identifiers (IRIs)", RFC 3987, January 2005.

[RFC4366] Blake-Wilson, S., Nystrom, M., Hopwood, D., Mikkelsen, J., and T. Wright, "Transport Layer Security (TLS) Extensions", RFC 4366, April 2006.

[RFC5234] Crocker, D. and P. Overell, "Augmented BNF for Syntax Specifications: ABNF", STD 68, RFC 5234, January 2008.

[WSAPI] Hickson, I., "The Web Sockets API", August 2010, <http://dev.w3.org/html5/websockets/>.

Author's Address

Adam Barth
Google, Inc.

Email: ietf@adambarth.com
URI: http://www.adambarth.com/