Network Working Group
Internet-Draft
Intended status: Standards Track
Expires: August 5, 2010

I. Hickson
Google, Inc.

February 1, 2010

The Web Socket protocol
draft-hixie-thewebsocketprotocol-71

Abstract

The Web Socket 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).

Author's note

This document is automatically generated from the same source document as the HTML5 specification. [HTML5]

Please send feedback to either the hybi@ietf.org list or the whatwg@whatwg.org list.

Status of this Memo

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

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF), its areas, and its working groups. Note that other groups may also distribute working documents as Internet-Drafts.

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."

The list of current Internet-Drafts can be accessed at http://www.ietf.org/ietf/1id-abstracts.txt.

The list of Internet-Draft Shadow Directories can be accessed at http://www.ietf.org/shadow.html.

This Internet-Draft will expire on August 5, 2010.

Copyright Notice

Copyright (c) 2010 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (http://trustee.ietf.org/license-info) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the BSD License.

Table of Contents

1. Introduction 4
1.1. Background 4
1.2. Protocol overview 4
1.3. Design philosophy 6
1.4. Security model 7
1.5. Relationship to TCP/IP and HTTP 7
1.6. Establishing a connection 8
1.7. Writing a simple Web Socket server 8
1.8. Subprotocols using the Web Socket protocol 9
2. Conformance requirements 11
2.1. Terminology 11
3. Web Socket URLs 12
3.1. Parsing Web Socket URLs 12
3.2. Constructing Web Socket URLs 13
4. Client-side requirements 14
4.1. Handshake 14
4.2. Data framing 22
4.3. Closing the connection 23
4.4. Handling errors in UTF-8 23
5. Server-side requirements 25
5.1. Sending the server's handshake 25
5.2. Reading the client's handshake 27
5.3. Data framing 28
5.4. Handling errors in UTF-8 29
6. Closing the connection 30
7. Security considerations 31
8. IANA considerations 32
8.1. Registration of ws: scheme 32
8.2. Registration of wss: scheme 33
8.3. Registration of the "WebSocket" HTTP Upgrade keyword . . . 34
8.4. WebSocket-Origin 34
8.5. WebSocket-Protocol 35
8.6. WebSocket-Location 36
9. Using the Web Socket protocol from other specifications . . . 37
10. Acknowledgements 38
11. Normative References 39
Author's Address 41

1. Introduction

1.1. Background

This section is non-normative.

Historically, creating an instant messenger chat client as a Web application has required an abuse of HTTP to poll the server for updates while sending upstream notifications as distinct HTTP calls.

This results in a variety of problems:

- The server is forced to use a number of different underlying TCP connections for each client: one for sending information to the client, and a new one for each incoming message.
- The wire protocol has a high overhead, with each client-to-server message having an HTTP header.
- The client-side script is forced to maintain a mapping from the outgoing connections to the incoming connection to track replies.

A simpler solution would be to use a single TCP connection for traffic in both directions. This is what the Web Socket protocol provides. Combined with the Web Socket API, it provides an alternative to HTTP polling for two-way communication from a Web page to a remote server. [WSAPI]

The same technique can be used for a variety of Web applications: games, stock tickers, multiuser applications with simultaneous editing, user interfaces exposing server-side services in real time, etc.

1.2. Protocol overview

This section is non-normative.

The protocol has two parts: a handshake, and then the data transfer.

The handshake from the client looks as follows:

```
GET /demo HTTP/1.1
Upgrade: WebSocket
Connection: Upgrade
Host: example.com
Origin: http://example.com
WebSocket-Protocol: sample
```

The handshake from the server looks as follows:

```
HTTP/1.1 101 Web Socket Protocol Handshake
Upgrade: WebSocket
Connection: Upgrade
WebSocket-Origin: http://example.com
WebSocket-Location: ws://example.com/demo
WebSocket-Protocol: sample
```

The first three lines in each case are hard-coded (the exact case and order matters); the remainder are an unordered ASCII case-insensitive set of fields, one per line, that match the following non-normative ABNF: [RFC5234]

```
  field         = 1*name-char colon [ space ] *any-char cr lf
  colon         = %x003A ; U+003A COLON (:)
  space         = %x0020 ; U+0020 SPACE
  cr            = %x000D ; U+000D CARRIAGE RETURN (CR)
  lf            = %x000A ; U+000A LINE FEED (LF)
  name-char     = %x0000-0009 / %x000B-000C / %x000E-0039 / %x003B-10FFFF
                  ; a Unicode character other than U+000A LINE FEED (LF),
U+000D CARRIAGE RETURN (CR), or U+003A COLON (:)
  any-char      = %x0000-0009 / %x000B-000C / %x000E-10FFFF
                  ; a Unicode character other than U+000A LINE FEED (LF) or
U+000D CARRIAGE RETURN (CR)
```

Lines that don't match the above production cause the connection to be aborted.

NOTE: The character set for the above ABNF is Unicode. The headers themselves are encoded as UTF-8.

Once the client and server have both sent their handshakes, and if the handshake was successful, then the data transfer part starts. This is a two-way communication channel where each side can, independently from the other, send data at will.

Data is sent in the form of UTF-8 text. Each frame of data starts with a 0x00 byte and ends with a 0xFF byte, with the UTF-8 text in between.

The Web Socket protocol uses this framing so that specifications that use the Web Socket protocol can expose such connections using an event-based mechanism instead of requiring users of those specifications to implement buffering and piecing together of messages manually.

The protocol is designed to support other frame types in future. Instead of the 0x00 byte, other bytes might in future be defined.

Frames denoted by bytes that do not have the high bit set (0x00 to

0x7F) are treated as described above (a stream of bytes terminated by 0xFF). Frames denoted by bytes that have the high bit set (0x80 to 0xFF) have a leading length indicator, which is encoded as a series of 7-bit bytes stored in octets with the 8th bit being set for all but the last byte. The remainder of the frame is then as much data as was specified.

This wire format for the data transfer part is described by the following non-normative ABNF, which is given in two alternative forms: the first describing the wire format as allowed by this specification, and the second describing how an arbitrary bytestream would be parsed. [RFC5234]

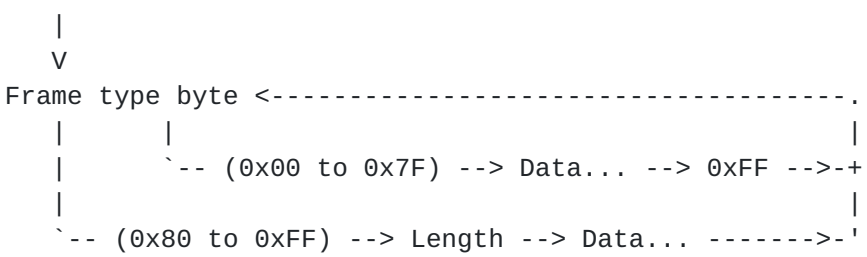
```
  ; the wire protocol as allowed by this specification
  frames        = *frame
  frame         = text-frame
  text-frame    = (%x00) *( UTF8-char ) %xFF

  ; the wire protocol including error-handling and forward-compatible
parsing rules
  frames        = *frame
  frame         = text-frame / binary-frame
  text-frame    = (%x00-%x7F) *( UTF8-char / %x80-%x7E ) %xFF
  binary-frame  = (%x80-%xFF) length < as many bytes as given by the length
>
  length        = *(%x80-%xFF) (%x00-%x7F)
```

The UTF8-char rule is defined in the UTF-8 specification. [RFC3629]

NOTE: The above ABNF is intended for a binary octet environment.

The following diagrams summarise the protocol:



1.3. Design philosophy

This section is non-normative.

The Web Socket protocol is designed on the principle that there should be minimal framing (the only framing that exists is to make

the protocol frame-based instead of stream-based, and to support a

distinction between Unicode text and binary frames). It is expected that metadata would be layered on top of Web Socket by the application layer, in the same way that metadata is layered on top of TCP/IP by the application layer (HTTP).

Conceptually, Web Socket is really just a layer on top of TCP/IP that adds a Web "origin"-based security model for browsers; adds an addressing and protocol naming mechanism to support multiple services on one port and multiple host names on one IP address; and layers a framing mechanism on top of TCP to get back to the IP packet mechanism that TCP is built on, but without length limits. Other than that, it adds nothing. Basically it is intended to be as close as possible to just exposing raw TCP/IP to script as possible given the constraints of the Web. It's also designed in such a way that its servers can share a port with HTTP servers, by having its handshake be a valid HTTP Upgrade handshake also.

1.4. Security model

This section is non-normative.

The Web Socket protocol uses the origin model used by Web browsers to restrict which Web pages can contact a Web Socket server when the Web Socket protocol is used from a Web page. Naturally, when the Web Socket protocol is used directly (not from a Web page), the origin model is not useful, as the client can provide any arbitrary origin string.

This protocol is intended to fail to establish a connection with servers of pre-existing protocols like SMTP or HTTP, while allowing HTTP servers to opt-in to supporting this protocol if desired. This is achieved by having a strict and elaborate handshake, and by limiting the data that can be inserted into the connection before the handshake is finished (thus limiting how much the server can be influenced).

1.5. Relationship to TCP/IP and HTTP

This section is non-normative.

The Web Socket protocol is an independent TCP-based protocol. Its only relationship to HTTP is that its handshake is interpreted by HTTP servers as an Upgrade request.

Based on the expert recommendation of the IANA, the Web Socket protocol by default uses port 80 for regular Web Socket connections and port 443 for Web Socket connections tunneled over TLS.

1.6. Establishing a connection

This section is non-normative.

There are several options for establishing a Web Socket connection.

On the face of it, the simplest method would seem to be to use port 80 to get a direct connection to a Web Socket server. Port 80 traffic, however, will often be intercepted by HTTP proxies, which can lead to the connection failing to be established.

The most reliable method, therefore, is to use TLS encryption and port 443 to connect directly to a Web Socket server. This has the advantage of being more secure; however, TLS encryption can be computationally expensive.

When a connection is to be made to a port that is shared by an HTTP server (a situation that is quite likely to occur with traffic to ports 80 and 443), the connection will appear to the HTTP server to be a regular GET request with an Upgrade offer. In relatively simple setups with just one IP address and a single server for all traffic to a single hostname, this might allow a practical way for systems based on the Web Socket protocol to be deployed. In more elaborate setups (e.g. with load balancers and multiple servers), a dedicated set of hosts for Web Socket connections separate from the HTTP servers is probably easier to manage.

1.7. Writing a simple Web Socket server

This section is non-normative.

If the Web Socket protocol is being used to provide a feature for a specific site, then the handshake can be hard-coded, and the data provided by the client in the handshake can be safely ignored. This section describes an implementation strategy for this case.

Listen on a port for TCP/IP. Upon receiving a connection request, open a connection and send the following bytes back to the client:

```
48 54 54 50 2F 31 2E 31  20 31 30 31 20 57 65 62
20 53 6F 63 6B 65 74 20  50 72 6F 74 6F 63 6F 6C
20 48 61 6E 64 73 68 61  6B 65 0D 0A 55 70 67 72
61 64 65 3A 20 57 65 62  53 6F 63 6B 65 74 0D 0A
43 6F 6E 6E 65 63 74 69  6F 6E 3A 20 55 70 67 72
61 64 65 0D 0A 57 65 62  53 6F 63 6B 65 74 2D 4F
72 69 67 69 6E 3A 20
```

Send the ASCII serialization of the origin from which the server is

willing to accept connections. [ORIGIN]

For example: |http://example.com|

Continue by sending the following bytes back to the client:

```
0D 0A 57 65 62 53 6F 63  6B 65 74 2D 4C 6F 63 61
74 69 6F 6E 3A 20
```

Send the URL of the Web Socket script.

For example: |ws://example.com/demo|

Finish the handshake by sending the four bytes 0x0D 0x0A 0x0D 0x0A to the client. Then, read data _from_ the client until four bytes 0x0D 0x0A 0x0D 0x0A are read.

NOTE: User agents will drop the connection after the handshake if the origin and URL sent as part of the algorithm above don't match what the client sent to the server, to protect the server from third-party scripts. This is why the server has to send these strings: to confirm which origins and URLs the server is willing to service.

At this point, there are two concerns: receiving frames and sending frames.

To receive a frame, read a byte, verify that it is a 0x00 byte, then read bytes until you find a 0xFF byte, and interpret all the bytes between the 0x00 and 0xFF bytes as a UTF-8 string (the frame payload, or message). This process can be repeated as necessary. If at any point the first byte of one of these sequences is not 0x00, then an error has occurred, and closing the connection is the appropriate response.

To send a frame, first send a 0x00 byte, then send the message as a UTF-8 string, then send a 0xFF byte. Again, this process can be repeated as necessary.

The connection can be closed as desired.

1.8. Subprotocols using the Web Socket protocol

This section is non-normative.

The client can request that the server use a specific subprotocol by including the |Websocket-Protocol| header in its handshake. If it is specified, the server needs to include the same header and value in its response for the connection to be established.

These subprotocol names do not need to be registered, but if a subprotocol is intended to be implemented by multiple independent Web Socket servers, potential clashes with the names of subprotocols defined independently can be avoided by using names that contain the domain name of the subprotocol's originator. For example, if Example Corporation were to create a Chat subprotocol to be implemented by many servers around the Web, they could name it "chat.example.com". If the Example Organisation called their competing subprotocol "example.org's chat protocol", then the two subprotocols could be implemented by servers simultaneously, with the server dynamically selecting which subprotocol to use based on the value sent by the client.

Subprotocols can be versioned in backwards-incompatible ways by changing the subprotocol name, eg. going from "bookings.example.net" to "bookings.example.net2". These subprotocols would be considered completely separate by Web Socket clients. Backwards-compatible versioning can be implemented by reusing the same subprotocol string but carefully designing the actual subprotocol to support this kind of extensibility.

2. Conformance requirements

All diagrams, examples, and notes in this specification are non-normative, as are all sections explicitly marked non-normative. Everything else in this specification is normative.

The key words "MUST", "MUST NOT", "REQUIRED", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in the normative parts of this document are to be interpreted as described in RFC2119. For readability, these words do not appear in all uppercase letters in this specification. [RFC2119]

Requirements phrased in the imperative as part of algorithms (such as "strip any leading space characters" or "return false and abort these steps") are to be interpreted with the meaning of the key word ("must", "should", "may", etc) used in introducing the algorithm.

Conformance requirements phrased as algorithms or specific steps may be implemented in any manner, so long as the end result is equivalent. (In particular, the algorithms defined in this specification are intended to be easy to follow, and not intended to be performant.)

Implementations may impose implementation-specific limits on otherwise unconstrained inputs, e.g. to prevent denial of service attacks, to guard against running out of memory, or to work around platform-specific limitations.

The conformance classes defined by this specification are user agents and servers.

2.1. Terminology

Converting a string to ASCII lowercase means replacing all characters in the range U+0041 to U+005A (i.e. LATIN CAPITAL LETTER A to LATIN CAPITAL LETTER Z) with the corresponding characters in the range U+0061 to U+007A (i.e. LATIN SMALL LETTER A to LATIN SMALL LETTER Z).

The term "URL" is used in this section in a manner consistent with the terminology used in HTML, namely, to denote a string that might or might not be a valid URI or IRI and to which certain error handling behaviors will be applied when the string is parsed. [HTML5]

3. Web Socket URLs

3.1. Parsing Web Socket URLs

The steps to *parse a Web Socket URL's components* from a string /url/ are as follows. These steps return either a /host/, a /port/, a /resource name/, and a /secure/ flag, or they fail.

1. If the /url/ string is not an absolute URL, then fail this algorithm. [WEBADDRESSES]

2. Resolve the /url/ string using the resolve a Web address algorithm defined by the Web addresses specification, with the URL character encoding set to UTF-8. [WEBADDRESSES] [RFC3629]

   NOTE: It doesn't matter what it is resolved relative to, since we already know it is an absolute URL at this point.

3. If /url/ does not have a <scheme> component whose value, when converted to ASCII lowercase, is either "ws" or "wss", then fail this algorithm.

4. If /url/ has a <fragment> component, then fail this algorithm.

5. If the <scheme> component of /url/ is "ws", set /secure/ to false; otherwise, the <scheme> component is "wss", set /secure/ to true.

6. Let /host/ be the value of the <host> component of /url/, converted to ASCII lowercase.

7. If /url/ has a <port> component, then let /port/ be that component's value; otherwise, there is no explicit /port/.

8. If there is no explicit /port/, then: if /secure/ is false, let /port/ be 80, otherwise let /port/ be 443.

9. Let /resource name/ be the value of the <path> component (which might be empty) of /url/.

10. If /resource name/ is the empty string, set it to a single character U+002F SOLIDUS (/).

11. If /url/ has a <query> component, then append a single U+003F QUESTION MARK character (?) to /resource name/, followed by the value of the <query> component.

12. Return /host/, /port/, /resource name/, and /secure/.

3.2. Constructing Web Socket URLs

The steps to *construct a Web Socket URL* from a /host/, a /port/, a /resource name/, and a /secure/ flag, are as follows:

1. Let /url/ be the empty string.

2. If the /secure/ flag is false, then append the string "ws://" to /url/. Otherwise, append the string "wss://" to /url/.

3. Append /host/ to /url/.

4. If the /secure/ flag is false and port is not 80, or if the /secure/ flag is true and port is not 443, then append the string ":" followed by /port/ to /url/.

5. Append /resource name/ to /url/.

6. Return /url/.

4. Client-side requirements

This section only applies to user agents, not to servers.

NOTE: This specification doesn't currently define a limit to the number of simultaneous connections that a client can establish to a server.

4.1. Handshake

When the user agent is to *establish a Web Socket 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/, and optionally with a string giving a /protocol/, it must run the following steps. The /host/ must be ASCII-only (i.e. it must have been punycode-encoded already if necessary). The /resource name/ and /protocol/ strings must be non-empty strings of ASCII characters in the range U+0020 to U+007E. The /resource name/ string must start with a U+002F SOLIDUS character (/) and must not contain a U+0020 SPACE character. [ORIGIN]

1. If the user agent already has a Web Socket 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.

   NOTE: This makes it harder for a script to perform a denial of service attack by just opening a large number of Web Socket connections to a remote host.

   NOTE: There is no limit to the number of established Web Socket 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.

2. _Connect_: If the user agent is configured to use a proxy when using the Web Socket protocol to connect to host /host/ and/or port /port/, then connect to that proxy and ask it to open a TCP/IP 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/IP 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 Web Socket connections separate from other proxies are encouraged to use a SOCKS proxy for Web Socket 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 Web Socket URL.

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

3. If the connection could not be opened, then fail the Web Socket connection and abort these steps.

4. 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 Web Socket 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]

5. Send the following bytes to the remote side (the server):

   ```
   47 45 54 20
   ```

   Send the /resource name/ value, encoded as ANSI_X3.4-1968 (US-ASCII). [RFC1345]

Send the following bytes:

```
20 48 54 54 50 2F 31 2E  31 0D 0A 55 70 67 72 61
64 65 3A 20 57 65 62 53  6F 63 6B 65 74 0D 0A 43
6F 6E 6E 65 63 74 69 6F  6E 3A 20 55 70 67 72 61
64 65 0D 0A
```

NOTE: The string "GET ", the path, " HTTP/1.1", CRLF, the string "Upgrade: WebSocket", CRLF, and the string "Connection: Upgrade", CRLF.

6. Send the following bytes:

```
48 6F 73 74 3A 20
```

Send the /host/ value, converted to ASCII lowercase, and encoded as ANSI_X3.4-1968 (US-ASCII). [RFC1345]

If /secure/ is false, and /port/ is not 80, or if /secure/ is true, and /port/ is not 443, then send an 0x3A byte (ASCII :) followed by the value of /port/, expressed as a base-ten integer, encoded as ANSI_X3.4-1968 (US-ASCII). [RFC1345]

Send the following bytes:

```
0D 0A
```

NOTE: The string "Host: ", the host, and CRLF.

7. Send the following bytes:

```
4F 72 69 67 69 6E 3A 20
```

Send the /origin/ value, converted to ASCII lowercase, encoded as ANSI_X3.4-1968 (US-ASCII). [ORIGIN] [RFC1345]

NOTE: The /origin/ value is a string that was passed to this algorithm.

Send the following bytes:

```
0D 0A
```

NOTE: The string "Origin: ", the origin, and CRLF.

8. If there is no /protocol/, then skip this step.

Otherwise, send the following bytes:

```
57 65 62 53 6F 63 6B 65  74 2D 50 72 6F 74 6F 63
6F 6C 3A 20
```

Send the /protocol/ value, encoded as ANSI_X3.4-1968 (US-ASCII). [RFC1345]

Send the following bytes:

```
0d 0a
```

NOTE: The string "WebSocket-Protocol: ", the protocol, and CRLF.

9. 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 HTTP headers that would be appropriate for that information should be sent at this point. [RFC2616] [RFC2109] [RFC2965]

   This includes "HttpOnly" cookies (cookies with the http-only-flag set to true); the Web Socket protocol is not considered a non-HTTP API.

   Each header must be on a line of its own (each ending with a CRLF sequence). For the purposes of this step, each header must not be split into multiple lines (despite HTTP otherwise allowing this with continuation lines).

10. Send the following bytes:

```
0d 0a
```

    NOTE: Just a CRLF (a blank line).

11. Read bytes from the server until either the connection closes, or a 0x0A byte is read. Let /header/ be these bytes, including the 0x0A byte.

    If /header/ is not at least two bytes long, or if the last two bytes aren't 0x0D and 0x0A respectively, then fail the Web Socket connection and abort these steps.

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

12. If /header/ consists of 44 bytes that exactly match the following, then let /mode/ be _normal_.

```
48 54 54 50 2F 31 2E 31  20 31 30 31 20 57 65 62
20 53 6F 63 6B 65 74 20  50 72 6F 74 6F 63 6F 6C
20 48 61 6E 64 73 68 61  6B 65 0D 0A
```

NOTE: The string "HTTP/1.1 101 Web Socket Protocol Handshake" followed by a CRLF pair.

NOTE: Note that this means that if a server responds with a Web Socket handshake but with the string "HTTP/1.0" or "HTTP/1.2" at the front, a normal Web Socket connection will not be established.

Otherwise, let /code/ be the substring of /header/ that starts from the byte after the first 0x20 byte, and ends with the byte before the second 0x20 byte. If there are not at least two 0x20 bytes in /header/, then fail the Web Socket connection and abort these steps.

If /code/, interpreted as ANSI_X3.4-1968 (US-ASCII), is "407", then either close the connection and jump back to step 2, providing appropriate authentication information, or fail the Web Socket connection. 407 is the code used by HTTP meaning "Proxy Authentication Required". User agents that support proxy authentication must interpret the response as defined by HTTP (e.g. to find and interpret the |Proxy-Authenticate| header). [RFC1345]

Otherwise, fail the Web Socket connection and abort these steps.

13. If /mode/ is _normal_, then read 41 bytes from the server.

If the connection closes before 41 bytes are received, or if the 41 bytes aren't exactly equal to the following bytes, then fail the Web Socket connection and abort these steps.

```
55 70 67 72 61 64 65 3A  20 57 65 62 53 6F 63 6B
65 74 0D 0A 43 6F 6E 6E  65 63 74 69 6F 6E 3A 20
55 70 67 72 61 64 65 0D  0A
```

NOTE: The string "Upgrade: WebSocket", CRLF, the string "Connection: Upgrade", CRLF.

User agents may apply a timeout to this step, failing the Web Socket connection if the server does not respond with the above bytes within a suitable time period.

14. Let /headers/ be a list of name-value pairs, initially empty.

15. _Header_: Let /name/ and /value/ be empty byte arrays.

16. Read a byte from the server.

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

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

    -> If the byte is 0x0D (ASCII CR)
       If the /name/ byte array is empty, then jump to the headers processing step. Otherwise, fail the Web Socket connection and abort these steps.

    -> If the byte is 0x0A (ASCII LF)
       Fail the Web Socket connection and abort these steps.

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

    -> If the byte is in the range 0x41 to 0x5A (ASCII 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 header name, terminated by a colon, converting upper-case ASCII letters to lowercase, and aborting if a stray CR or LF is found.

17. Read a byte from the server.

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

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

    -> If the byte is 0x20 (ASCII space)
       Ignore the byte and move on to the next step.

    -> Otherwise
       Treat the byte as described by the list in the next step, then move on to that next step for real.

NOTE: This skips past a space character after the colon, if necessary.

18. Read a byte from the server.

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

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

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

    -> If the byte is 0x0A (ASCII LF)
       Fail the Web Socket 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 header value, terminated by a CRLF.

19. Read a byte from the server.

    If the connection closes before this byte is received, or if the byte is not a 0x0A byte (ASCII LF), then fail the Web Socket connection and abort these steps.

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

20. Append an entry to the /headers/ 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.

21. Return to the "Header" step above.

22. _Headers 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 (ASCII LF), then fail the Web Socket connection and abort these steps.

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

23. If /mode/ is _normal_, then: If there is not exactly one entry in the /headers/ list whose name is "websocket-origin", or if there is not exactly one entry in the /headers/ list whose name is "websocket-location", or if the /protocol/ was specified but there is not exactly one entry in the /headers/ list whose name is "websocket-protocol", or if there are any entries in the /headers/ list whose names are the empty string, then fail the Web Socket connection and abort these steps. Otherwise, handle each entry in the /headers/ list as follows:

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

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

    -> If the entry's name is "websocket-protocol"
       If there was a /protocol/ specified, and the value is not exactly equal to /protocol/, then fail the Web Socket connection and abort these steps. (If no /protocol/ was specified, the header is ignored.)

    -> If the entry's name is "set-cookie" or "set-cookie2" or another cookie-related header name
       Handle the cookie as defined by the appropriate specification, 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]

    -> Any other name
       Ignore it.

24. The *Web Socket connection is established*. Now the user agent must send and receive to and from the connection as described in the next section.

4.2. Data framing

Once a Web Socket connection is established, the user agent must run through the following state machine for the bytes sent by the server.

1. Try to read a byte from the server. Let /frame type/ be that byte.

   If no byte could be read because the Web Socket connection is closed, then abort.

2. Handle the /frame type/ byte as follows:

   If the high-order bit of the /frame type/ byte is set (i.e. if /frame type/ _and_ed with 0x80 returns 0x80)
   Run these steps. If at any point during these steps a read is attempted but fails because the Web Socket connection is closed, then abort.

   1. Let /length/ be zero.

   2. _Length_: Read a byte, let /b/ be that byte.

   3. Let /b_v/ be integer corresponding to the low 7 bits of /b/ (the value you would get by _and_ing /b/ with 0x7F).

   4. Multiply /length/ by 128, add /b_v/ to that result, and store the final result in /length/.

   5. If the high-order bit of /b/ is set (i.e. if /b/ _and_ed with 0x80 returns 0x80), then return to the step above labeled _length_.

   6. Read /length/ bytes.

   7. Discard the read bytes.

   If the high-order bit of the /frame type/ byte is _not_ set (i.e. if /frame type/ _and_ed with 0x80 returns 0x00)
   Run these steps. If at any point during these steps a read is attempted but fails because the Web Socket connection is closed, then abort.

   1. Let /raw data/ be an empty byte array.

   2. _Data_: Read a byte, let /b/ be that byte. If the client runs out of resources for buffering the incoming data, or hits an artificial resource limit intended to avoid

resource starvation, then it must fail the Web Socket connection and abort these steps.

3. If /b/ is not 0xFF, then append /b/ to /raw data/ and return to the previous step (labeled _data_).

4. Interpret /raw data/ as a UTF-8 string, and store that string in /data/.

5. If /frame type/ is 0x00, then *a message has been received* with text /data/. Otherwise, discard the data.

3. Return to the first step to read the next byte.

If the user agent is faced with content that is too large to be handled appropriately, then it must fail the Web Socket connection.

Once a Web Socket connection is established, the user agent must use the following steps to *send /data/ using the Web Socket*:

1. Send a 0x00 byte to the server.

2. Encode /data/ using UTF-8 and send the resulting byte stream to the server.

3. Send a 0xFF byte to the server.

If at any point there is a fatal problem with sending data to the server, the user agent must fail the Web Socket connection.

4.3. Closing the connection

To *fail the Web Socket connection*, the user agent must close the Web Socket connection, and may report the problem to the user (which would be especially useful for developers). However, user agents must not convey the failure information to the script that attempted the connection in a way distinguishable from the Web Socket being closed normally.

Except as indicated above or as specified by the application layer (e.g. a script using the Web Socket API), user agents should not close the connection.

4.4. Handling errors in UTF-8

When a client is to interpret a byte stream as UTF-8 but finds that the byte stream is not in fact a valid UTF-8 stream, then any bytes

or sequences of bytes that are not valid UTF-8 sequences must be interpreted as a U+FFFD REPLACEMENT CHARACTER.

5. Server-side requirements

This section only applies to servers.

5.1. Sending the server's handshake

When a client establishes a Web Socket connection to a server, the server must either close the connection or send the server handshake. Servers may read part or all of the client's handshake (as described in the next section) before closing the connection or sending all of their side of the handshake; indeed, in some cases this is necessary as the server might need to use some of the information in the client's handshake to construct it's own handshake.

If the server supports encryption, then the server must perform a TLS handshake over the connection before sending the server handshake. 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 the server must close the connection; otherwise, all further communication for the connection (including the server handshake) must run through the encrypted tunnel. [RFC2246]

To send the server handshake, the server must first establish the following information:

/origin/
: The ASCII serialization of the origin that the server is willing to communicate with. 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]

/host/
: The host name or IP address of the Web Socket 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.

/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 client's handshake, specifically from the "Host"

field.

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

/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 from the "WebSocket-Protocol" field. 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.

Having established this information, the server must start the handshake. The initial part of the server's handshake is invariant, and must consist of the following bytes:

```
48 54 54 50 2F 31 2E 31  20 31 30 31 20 57 65 62
20 53 6F 63 6B 65 74 20  50 72 6F 74 6F 63 6F 6C
20 48 61 6E 64 73 68 61  6B 65 0D 0A 55 70 67 72
61 64 65 3A 20 57 65 62  53 6F 63 6B 65 74 0D 0A
43 6F 6E 6E 65 63 74 69  6F 6E 3A 20 55 70 67 72
61 64 65 0D 0A 57 65 62  53 6F 63 6B 65 74 2D 4F
72 69 67 69 6E 3A 20
```

These bytes must be the first bytes sent on the TCP connection by the server. They must be followed by the /origin/ string, encoded as ANSI_X3.4-1968 (US-ASCII), followed by the following bytes: [RFC1345]

```
0D 0A 57 65 62 53 6F 63  6B 65 74 2D 4C 6F 63 61
74 69 6F 6E 3A 20
```

The server must then send the string that results from constructing a Web Socket URL from /host/, /port/, /resource name/, and /secure flag/, encoded as ANSI_X3.4-1968 (US-ASCII). [RFC1345]

If the /subprotocol/ is not null, then the server must then send the following bytes:

```
0D 0A 57 65 62 53 6F 63  6B 65 74 2D 50 72 6F 74
6F 63 6F 6C 3A 20
```

...followed by the /subprotocol/ string, encoded as ANSI_X3.4-1968 (US-ASCII). [RFC1345]

Finally, the server must end its side of the handshake by sending the four bytes 0x0D 0x0A 0x0D 0x0A to the client.

5.2. Reading the client's handshake

When a client starts a Web Socket connection, it sends its part of the handshake. This consists of a number of fields separated by CRLF pairs (bytes 0x0D 0x0A).

The first field consists of three tokens separated by space characters (byte 0x20). The first token is the string "GET", the middle token is the resource name, and the third is the string "HTTP/1.1".

If the first field does not have three tokens, or if the first and third tokens aren't the strings given in the previous paragraph, or if the second token doesn't begin with U+002F SOLIDUS character (/), the server should abort the connection: it either represents an errorneous Web Socket client or a connection from a client expecting another protocol altogether.

The subsequent fields consist of a string representing a name, a colon and a space (bytes 0x3A 0x20), and a string representing a value. The possible names, and the meaning of their corresponding values, are as follows:

Upgrade (bytes 55 70 67 72 61 64 65; always the first name-value pair)
: Invariant part of the handshake. Will always have a value consisting of bytes 57 65 62 53 6F 63 6B 65 74.

Connection (bytes 43 6F 6E 6E 65 63 74 69 6F 6E; always the second name-value pair)
: Invariant part of the handshake. Will always have a value consisting of bytes 55 70 67 72 61 64 65.

Host (bytes 48 6F 73 74; always the third name-value pair)
: The value gives the hostname that the client intended to use when opening the Web Socket. 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. The value must be interpreted as UTF-8.

Origin (bytes 4F 72 69 67 69 6E; always the fourth name-value pair)
: 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 Web Socket. 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. The value must be interpreted as UTF-8.

WebSocket-Protocol (bytes 57 65 62 53 6F 63 6B 65 74 2D 50 72 6F 74 6F 63 6F 6C; optional, if present, will be the fifth name-value pair)

The value gives the name of a subprotocol that the client is intending to select. It would be interesting if the server supports multiple protocols or protocol versions. The value must be interpreted as UTF-8.

Other fields

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.

A final field consisting of the empty string (two consecutive CRLF pairs) indicates the end of the client's handshake.

Any fields that lack the colon-space separator must at a minimum be discarded and may cause the server to disconnect.

Whether the server does or does not read the client handshake, it must at a minimum read (and optionally discard) bytes until it has read the first sequence of four bytes 0x0A 0x0D 0x0A 0x0D, which signals the end of the client handshake. Servers may do this before or after sending their handshake, but must do it before reading frames from the client as described in the next section.

5.3. Data framing

The server must run through the following steps to process the bytes sent by the client:

1. _Frame_: Read a byte from the client. Let /type/ be that byte.

2. If /type/ is not a 0x00 byte, then the server may disconnect from the client.

3. If the most significant bit of /type/ is not set, then run the following steps:

    1. Let /raw data/ be an empty byte array.

    2. _Data_: Read a byte, let /b/ be that byte.

    3. If /b/ is not 0xFF, then append /b/ to /raw data/ and return to the previous step (labeled _data_).

    4. Interpret /raw data/ as a UTF-8 string, and apply whatever server-specific processing is to occur for the resulting

string (the message from the client).

Otherwise, the most significant bit of /type/ is set. Run the following steps. This can never happen if /type/ is 0x00, and therefore these steps are not necessary if the server aborts when /type/ is not 0x00, as allowed above.

5. Let /length/ be zero.

6. _Length_: Read a byte, let /b/ be that byte.

7. Let /b_v/ be integer corresponding to the low 7 bits of /b/ (the value you would get by _and_ing /b/ with 0x7F).

8. Multiply /length/ by 128, add /b_v/ to that result, and store the final result in /length/.

9. If the high-order bit of /b/ is set (i.e. if /b/ _and_ed with 0x80 returns 0x80), then return to the step above labeled _length_.

10. Read /length/ bytes.

11. Discard the read bytes.

4. Return to the step labeled _frame_.

The server must run through the following steps to send strings to the client:

1. Send a 0x00 byte to the client to indicate the start of a string.

2. Encode /data/ using UTF-8 and send the resulting byte stream to the client.

3. Send a 0xFF byte to the client to indicate the end of the message.

5.4. Handling errors in UTF-8

When a server is to interpret a byte stream as UTF-8 but finds that the byte stream is not in fact a valid UTF-8 stream, behaviour is undefined. A server could close the connection, convert invalid byte sequences to U+FFFD REPLACEMENT CHARACTERs, store the data verbatim, or perform application-specific processing. Subprotocols layered on the Web Socket protocol might define specific behavior for servers.

6. Closing the connection

To *close the Web Socket connection*, either the user agent or the server closes the TCP/IP connection. There is no closing handshake. When a user agent notices that the server has closed its connection, it must immediately close its side of the connection also. Whether the user agent or the server closes the connection first, it is said that the *Web Socket connection is closed*.

Servers may close the Web Socket connection whenever desired. User agents should not close the Web Socket connection arbitrarily.

7. Security considerations

While this protocol is intended to be used by scripts in Web pages, it can also be used directly by hosts. Such hosts are acting on their own behalf, and can therefore send fake "Origin" headers, misleading the server. Servers should therefore be careful about assuming that they are talking directly to scripts from known origins, and must consider that they might be accessed in unexpected ways. In particular, a server should not trust that any input is valid.

EXAMPLE: For example, if the server uses input as part of SQL queries, all input text should be escaped before being passed to the SQL server, lest the server be susceptible to SQL injection.

Servers that are not intended to process input from any Web page but only for certain sites should verify the "Origin" header is an origin they expect, and should only respond with the corresponding "WebSocket-Origin" if it is an accepted origin. Servers that only accept input from one origin can just send back that value in the "WebSocket-Origin" header, without bothering to check the client's value.

If at any time a server is faced with data that it does not understand, or that violates some criteria by which the server determines safety of input, or when the server sees a handshake that does not correspond to the values the server is expecting (e.g. incorrect path or origin), the server should just disconnect. It is always safe to disconnect.

The biggest security risk when sending text data using this protocol is sending data using the wrong encoding. If an attacker can trick the server into sending data encoded as ISO-8859-1 verbatim (for instance), rather than encoded as UTF-8, then the attacker could inject arbitrary frames into the data stream.

8. IANA considerations

8.1. Registration of ws: scheme

A |ws:| URL identifies a Web Socket server and resource name.

URI scheme name.
ws

Status.
Permanent.

URI scheme syntax.
In ABNF terms using the terminals from the URI specifications: [RFC5234] [RFC3986]

```
"ws" ":" hier-part [ "?" query ]
```

The path and query components form the resource name sent to the server to identify the kind of service desired. Other components have the meanings described in RFC3986.

URI scheme semantics.
The only operation for this scheme is to open a connection using the Web Socket protocol.

Encoding considerations.
Characters in the host component that are excluded by the syntax defined above must be converted from Unicode to ASCII by applying the IDNA ToASCII algorithm to the Unicode host name, with both the AllowUnassigned and UseSTD3ASCIIRules flags set, and using the result of this algorithm as the host in the URI. [RFC3490]

Characters in other components that are excluded by the syntax defined above must be converted from Unicode to ASCII by first encoding the characters as UTF-8 and then replacing the corresponding bytes using their percent-encoded form as defined in the URI and IRI specification. [RFC3986] [RFC3987]

Applications/protocols that use this URI scheme name.
Web Socket protocol.

Interoperability considerations.
None.

Security considerations.
   See "Security considerations" section above.

Contact.
   Ian Hickson <ian@hixie.ch>

Author/Change controller.
   Ian Hickson <ian@hixie.ch>

References.
   This document.

8.2. Registration of wss: scheme

A |wss:| URL identifies a Web Socket server and resource name, and indicates that traffic over that connection is to be encrypted.

URI scheme name.
   wss

Status.
   Permanent.

URI scheme syntax.
   In ABNF terms using the terminals from the URI specifications: [RFC5234] [RFC3986]

```
"wss" ":" hier-part [ "?" query ]
```

   The path and query components form the resource name sent to the server to identify the kind of service desired. Other components have the meanings described in RFC3986.

URI scheme semantics.
   The only operation for this scheme is to open a connection using the Web Socket protocol, encrypted using TLS.

Encoding considerations.
   Characters in the host component that are excluded by the syntax defined above must be converted from Unicode to ASCII by applying the IDNA ToASCII algorithm to the Unicode host name, with both the AllowUnassigned and UseSTD3ASCIIRules flags set, and using the result of this algorithm as the host in the URI. [RFC3490]

   Characters in other components that are excluded by the syntax defined above must be converted from Unicode to ASCII by first encoding the characters as UTF-8 and then replacing the corresponding bytes using their percent-encoded form as defined in

the URI and IRI specification. [RFC3986] [RFC3987]

Applications/protocols that use this URI scheme name.
Web Socket protocol over TLS.

Interoperability considerations.
None.

Security considerations.
See "Security considerations" section above.

Contact.
Ian Hickson <ian@hixie.ch>

Author/Change controller.
Ian Hickson <ian@hixie.ch>

References.
This document.

8.3. Registration of the "WebSocket" HTTP Upgrade keyword

Name of token.
WebSocket

Author/Change controller.
Ian Hickson <ian@hixie.ch>

Contact.
Ian Hickson <ian@hixie.ch>

References.
This document.

8.4. WebSocket-Origin

This section describes a header field for registration in the Permanent Message Header Field Registry. [RFC3864]

Header field name
WebSocket-Origin

Applicable protocol
http

Status
reserved; do not use outside Web Socket handshake

Author/Change controller
IETF

Specification document(s)
This document is the relevant specification.

Related information
None.

The |WebSocket-Origin| header is used in the Web Socket handshake. It is sent from the server to the client to confirm the origin of the script that opened the connection. This enables user agents to verify that the server is willing to serve the script that opened the connection.

8.5. WebSocket-Protocol

This section describes a header field for registration in the Permanent Message Header Field Registry. [RFC3864]

Header field name
WebSocket-Protocol

Applicable protocol
http

Status
reserved; do not use outside Web Socket handshake

Author/Change controller
IETF

Specification document(s)
This document is the relevant specification.

Related information
None.

The |WebSocket-Protocol| header is used in the Web Socket handshake. It is sent from the client to the server and back from the server to the client to confirm the subprotocol of the connection. This enables scripts to both select a subprotocol and be sure that the server agreed to serve that subprotocol.

8.6. WebSocket-Location

This section describes a header field for registration in the Permanent Message Header Field Registry. [RFC3864]

Header field name
WebSocket-Location

Applicable protocol
http

Status
reserved; do not use outside Web Socket handshake

Author/Change controller
IETF

Specification document(s)
This document is the relevant specification.

Related information
None.

The |WebSocket-Location| header is used in the Web Socket handshake. It is sent from the server to the client to confirm the URL of the connection. This enables the client to verify that the connection was established to the right server, port, and path, instead of relying on the server to verify that the requested host, port, and path are correct.

9. Using the Web Socket protocol from other specifications

The Web Socket protocol is intended to be used by another specification to provide a generic mechanism for dynamic author-defined content, e.g. in a specification defining a scripted API.

Such a specification first needs to "establish a Web Socket connection", providing that algorithm with:

- The destination, consisting of a /host/ and a /port/.
- A /resource name/, which allows for multiple services to be identified at one host and port.
- A /secure/ flag, which is true if the connection is to be encrypted, and false otherwise.
- An ASCII serialization of an origin that is being made responsible for the connection. [ORIGIN]
- Optionally a string identifying a protocol that is to be layered over the Web Socket connection.

The /host/, /port/, /resource name/, and /secure/ flag are usually obtained from a URL using the steps to parse a Web Socket URL's components. These steps fail if the URL does not specify a Web Socket.

If a connection can be established, then it is said that the "Web Socket connection is established".

If at any time the connection is to be closed, then the specification needs to use the "close the Web Socket connection" algorithm.

When the connection is closed, for any reason including failure to establish the connection in the first place, it is said that the "Web Socket connection is closed".

While a connection is open, the specification will need to handle the cases when "a Web Socket message has been received" with text /data/.

To send some text /data/ to an open connection, the specification needs to "send /data/ using the Web Socket".

10. Acknowledgements

The Web Socket protocol is the result of many years of development, and as such hundreds of people have contributed to the specification during its lifetime. Unfortunately, since the specification started as nothing but a minor section of the larger WHATWG Web Applications 1.0 specification, and later the HTML5 specification, no record was kept of who exactly contributed to what ended up becoming this specification as opposed to who contributed to other parts of that document.

The reader is therefore referred to the Acknowledgements section of the HTML5 specification for a full list of all contributions that have been made to the source document from which this specification is generated. [HTML5]

11. Normative References

[HTML5] Hickson, I., "HTML5", February 2010.

[ORIGIN] Barth, A., Jackson, C., and I. Hickson, "The HTTP Origin Header", September 2009.

[RFC1345] Simonsen, K., "Character Mnemonics and Character Sets", RFC 1345, June 1992.

[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.

[WEBADDRESSES] Connolly, D. and C. Sperberg-McQueen, "Web addresses in HTML 5", May 2009.

[WSAPI] Hickson, I., "The Web Sockets API", February 2010.

Author's Address

Ian Hickson
Google, Inc.

Email: ian@hixie.ch
URI: http://ln.hixie.ch/