Network Working Group   I. Hickson
Internet-Draft   Google, Inc.
Intended status: Standards Track   October 13, 2009
Expires: April 16, 2010

The Web Socket protocol
draft-hixie-thewebsocketprotocol-47

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 April 16, 2010.

Copyright Notice

Abstract

This protocol enables two-way communication between a user agent running untrusted code running in a controlled environment to a remote host that understands the protocol. It is designed to be easy to implement on the server side.

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.

Table of Contents

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:

o 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.

o The wire protocol has a high overhead, with each client-to-server message having an HTTP header.

o 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
```

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.

The following diagrams summarise the protocol:

```
Handshake
   |
  \|/
Frame type byte <--------------------------------------.
   |      |                                            |
   |      `-- (0x00 .. 0x7F) --> Data... --> 0xFF -->-+
   |                                                   |
   `-- (0x80 .. 0xFF) --> Length --> Data... ------->-'
```

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.

The simplest method is 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 second simplest method 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.

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 .. U+005A (i.e. LATIN CAPITAL LETTER A to LATIN CAPITAL LETTER Z) with the corresponding characters in the range U+0061 .. 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 /protocol/ is specified but is either the empty string or contains characters that are not in the range U+0021 .. U+007E, then fail this algorithm.

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

3. 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.

4. If /url/ does not have a <scheme> component whose value is either "ws" or "wss", when compared in an ASCII case-insensitive manner, 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 /resource name/ string must start with a U+002F SOLIDUS 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 WebSocket 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]

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

```
47 45 54 20
```

Send the /resource name/ value, encoded as US-ASCII.

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 US-ASCII.

   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 US-ASCII.

   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 US-ASCII. [ORIGIN]

   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 US-ASCII.

   Send the following bytes:

   0d 0a

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

9. If the client has any authentication information or 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]

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

   EXAMPLE: For example, if the server had a username and password that applied to |http://example.com/socket|, and the Web Socket was being opened to |ws://example.com/socket|, it could send them:

   ```
   Authorization: Basic d2FsbGU6ZXZl
   ```

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.

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 ASCII, is "401", then let /mode/ be _authenticate_.

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.

NOTE: This step is skipped if /mode/ is _authenticate_.

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 .. 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.

If /mode/ is _authenticate_, then: If there is not exactly one entry in the /headers/ list whose name is "www-authenticate", 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 "www-authenticate"
Obtain credentials in a manner consistent with the requirements for handling the |WWW-Authenticate| header in HTTP, and then close the connection (if the server has not already done so) and jump back to the step labeled _connect_, including the relevant authentication headers in the new request. [RFC2616]

-> 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. Minimal handshake

NOTE: This section describes the minimal requirements for a server-side implementation of Web Sockets.

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
```

Send the string "WebSocket-Origin" followed by a U+003A COLON (:) and a U+0020 SPACE, followed by the ASCII serialization of the origin from which the server is willing to accept connections, followed by a CRLF pair (0x0D 0x0A). [ORIGIN]

For instance:

```
WebSocket-Origin: http://example.com
```

Send the string "WebSocket-Location" followed by a U+003A COLON (:) and a U+0020 SPACE, followed by the URL of the Web Socket script, followed by a CRLF pair (0x0D 0x0A).

For instance:

```
WebSocket-Location: ws://example.com/demo
```

NOTE: Do not include the port if it is the default port for Web Socket protocol connections of the type in question (80 for unencrypted connections and 443 for encrypted connections).

Send another CRLF pair (0x0D 0x0A).

Read data from the client until four bytes 0x0D 0x0A 0x0D 0x0A are read. This data must either be discarded or handled as described in the following section describing the handshake details.

If the connection isn't dropped at this point, go to the data framing section.

5.2. Handshake details

The previous section ignores the data that is transmitted by the client during the handshake.

The data sent by the client consists of a number of fields separated by CR LF pairs (bytes 0x0D 0x0A).

The first field consists of three tokens separated by space characters (byte 0x20). The middle token is the path being opened. If the server supports multiple paths, then the server should echo the value of this field in the initial handshake, as part of the URL given on the |WebSocket-Location| line (after the appropriate scheme and host).

If the first field does not have three tokens, the server should abort the connection as it probably represents an errorneous client.

The remaining fields consist of name-value pairs, with the name part separated from the value part by a colon and a space (bytes 0x3A 0x20). Of these, several are interesting:

Host (bytes 48 6F 73 74)
: 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 right host has to be output as part of the URL given on the |WebSocket-Location| line of the handshake described above, to verify that the server knows that it is really representing that host.

Origin (bytes 4F 72 69 67 69 6E)
: 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.

  If the server supports connections from more than one origin, then the server should echo the value of this field in the initial handshake, on the |WebSocket-Origin| line.

Other fields
Other fields can be used, such as "Cookie" or "Authorization", for authentication purposes.

Any fields that lack the colon-space separator should be discarded and may cause the server to disconnect.

5.3. Data framing

NOTE: This section only describes how to handle content that this specification allows user agents to send (text). It doesn't handle any arbitrary content in the same way that the requirements on user agents defined earlier handle any content including possible future extensions to the protocols.

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

1. Read a byte from the client. Assuming everything is going according to plan, it will be a 0x00 byte. If the byte is not a 0x00 byte, then the server may disconnect.

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

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

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

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

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

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.

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. Whether the user agent or the server closes the connection, it is said that the *Web Socket connection is closed*.

When a user agent is to close the Web Socket connection, it must drop all subsequent data from the server and must act as if the server had immediately closed its side of the connection.

When a user agent notices that the Web Socket connection is closed, it must immediately close its side of the connection.

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.

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 WebSocket handshake

Author/Change controller
IETF

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

Related information
None.

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 WebSocket handshake

Author/Change controller
IETF

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

Related information
None.

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 WebSocket handshake

Author/Change controller
IETF

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

Related information
None.

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:

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

[HTML5] Hickson, I., "HTML5", October 2009.

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

[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.

[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", October 2009.

Author's Address

Ian Hickson
Google, Inc.

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