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

I. Hickson
Google, Inc.

August 4, 2009

The Web Socket protocol
draft-hixie-thewebsocketprotocol-29

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 February 5, 2010.

Copyright Notice

Copyright (c) 2009 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 in effect on the date of publication of this document (http://trustee.ietf.org/license-info). Please review these documents carefully, as they describe your rights and restrictions with respect to this document.

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 intended to fail to communicate with servers of pre-existing protocols like SMTP or HTTP, while allowing HTTP servers to opt-in to supporting this protocol if desired. 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 4
1.1. Security model 4
1.2. Relationship to TCP/IP and HTTP 4
1.3. Establishing a connection 5
1.4. The Web Socket protocol 5
2. Conformance requirements 7
3. Client-side requirements 8
3.1. Handshake 8
3.2. Data framing 17
3.3. Handling errors in UTF-8 18
4. Server-side requirements 19
4.1. Minimal handshake 19
4.2. Handshake details 20
4.3. Data framing 21
5. Closing the connection 22
6. Security considerations 23
7. IANA considerations 24
8. Normative References 25
Author's Address 26

1. Introduction

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.

1.1. 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.

1.2. 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.

The Web Socket protocol by default uses port 81 for regular Web Socket connections and port 815 for Web Socket connections tunneled over TLS.

1.3. 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 81 to get a direct connection to a Web Socket server. However, this port may be blocked by firewalls.

The second simplest method is to use TLS encryption and port 815 to connect directly to a Web Socket server. This is the preferred solution, as it is secure and correct. However, TLS encryption can be computationally expensive, and port 815 might also be blocked by firewalls.

To avoid firewalls, ports 80 and 443 might be used instead. These are the HTTP and HTTPS ports. Port 80 traffic, however, will often be intercepted by HTTP proxies, which can lead to the connection failing to be established.

Port 443, using encryption, is therefore the most reliable solution. It is unlikely to be blocked by a firewall or intercepted by a proxy. However, again, 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.

1.4. The Web Socket protocol

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.

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.

3. 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.

3.1. Handshake

When the user agent is to *establish a Web Socket connection* to a host /host/, optionally on port /port/, from an origin /origin/, with a flag /secure/, with a particular /resource name/, and optionally with a particular /protocol/, it must run the following steps.

NOTE: The /host/ and /origin/ strings will be all-lowercase when this algorithm is invoked.

1. If there is no explicit /port/, then: if /secure/ is false, let /port/ be 81, otherwise let /port/ be 815.

2. 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.

3. _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.

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

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

6. 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.

7. Send the following bytes:

48 6f 73 74 3a 20

Send the /host/ value, encoded as US-ASCII.

Send the following bytes:

0d 0a

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

8. Send the following bytes:

4f 72 69 67 69 6e 3a 20

Send the /origin/ value, encoded as US-ASCII.

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.

9. 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.

10. 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:80/socket|, it could send them:

Authorization: Basic d2FsbGU6ZXZl

However, it would not send them if the Web Socket was being opened to |ws://example.com/socket|, as that uses a different port (81, not 80).

11. Send the following bytes:

0d 0a

NOTE: Just a CRLF (a blank line).

12. 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.

13. 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 either "301", "302", "303", "307", then let /mode/ be _redirect_.

If /code/, interpreted as ASCII, is "401", then let /mode/ be _authenticate_.

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

14. 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 _redirect_ or _authenticate_.

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

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

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 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.

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 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.

19. 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.

20. 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.

21. 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.

22. Return to the "Header" step above.

23. _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.

24. 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.

-> If the entry's name is "websocket-location"
   If the value is not exactly equal to a string consisting of the following components in the same order, then fail the Web Socket connection and abort these steps:

   1. The string "ws" if /secure/ is false and "wss" if /secure/ is true

   2. The three characters "://".

   3. The value of /host/.

   4. If /secure/ is false and /port/ is not 81, or if /secure/ is true and /port/ is not 815: a ":" character followed by the value of /port/.

   5. The value of /resource name/.

-> 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 spec, 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 _redirect_, then: If there is not exactly one entry in the /headers/ list whose name is "location", 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 "location"
   Parse the value of the entry. If this fails, or if the parsed URL does not have a <scheme> component whose value is either "ws" or "wss", when compared in an ASCII case-insensitive manner, then fail the Web Socket connection and abort these steps.

If /secure/ is true but the newly parsed URL's <scheme> component is an ASCII case-insensitive match for "ws", then fail the Web Socket connection and abort these steps. Otherwise, if /secure/ is false but the newly parsed URL's <scheme> component is an ASCII case-insensitive match for "wss", then set /secure/ to true.

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

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

If there is an explicit /port/ and the user agent is configured to block access to that port, then fail the Web Socket connection and abort these steps.

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

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

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.

Close the connection if the server has not already done so.

Jump back to the first step of this algorithm.

-> 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.

25. 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.

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.

3.2. 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.

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.

3.3. 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.

4. Server-side requirements

This section only applies to servers.

4.1. 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 (":") followed by the ASCII serialization of the origin from which the server is willing to accept connections, followed by a CRLF pair (0x0d 0x0a).

For instance:

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

Send the string "WebSocket-Location" followed by a U+003A COLON (":") followed by the URL of the Web Socket script, followed by a CRLF pair (0x0d 0x0a).

For instance:

```
WebSocket-Location: ws://example.com:80/demo
```

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.

4.2. 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.

4.3. 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.

5. 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*.

Servers may close the Web Socket connection whenever desired.

User agents should not close the Web Socket connection arbitrarily.

6. Security considerations

** ISSUE ** ...

7. IANA considerations

** ISSUE ** ...(two URI schemes, two ports, HTTP Upgrade keyword)

8. Normative References

[HTML5] Hickson, I., "HTML5", August 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.

Author's Address

Ian Hickson
Google, Inc.

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