The WebSocket protocol
draft-ietf-hybi-thewebsocketprotocol-01

Abstract

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

Note

This draft is meant to reflect changes in direction in the HyBi working group. There is not yet consensus on everything in this draft. Specifically, details about the framing are still under discussion, however this draft is much closer to what the group is discussing than the previous draft. There have also been proposals to change the handshake, so the handshake is also not in a final form.

Status of this Memo

Table of Contents

1. Introduction

[TOC](#)

1.1. Background

[TOC](#)

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 WebSocket protocol provides.
Combined with the WebSocket API, it provides an alternative to HTTP

polling for two-way communication from a Web page to a remote server.
[WSAPI] (Hickson, I., "The Web Sockets API," August 2010.)
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
Host: example.com
Connection: Upgrade
Sec-WebSocket-Key2: 12998 5 Y3 1  .P00
Sec-WebSocket-Protocol: sample
Upgrade: WebSocket
Sec-WebSocket-Key1: 4 @1  46546xW%0l 1 5
Origin: http://example.com

^n:ds[4U
```

The handshake from the server looks as follows:

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

8jKS'y:G*Co,Wxa-
```

The leading line from the client follows the Request-Line format. The
leading line from the server follows the Status-Line format. The
Request-Line and Status-Line productions are defined in the HTTP
specification.
After the leading line in both cases come an unordered ASCII case-
insensitive set of fields, one per line, that each match the following
non-normative ABNF: [RFC5234] (Crocker, D. and P. Overell, "Augmented
BNF for Syntax Specifications: ABNF," January 2008.)

```
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)
```

NOTE: The character set for the above ABNF is Unicode. The fields
themselves are encoded as UTF-8.
Lines that don't match the above production cause the connection to be
aborted.
Finally, after the last field, the client sends 10 bytes starting with
0x0D 0x0A and followed by 8 random bytes, part of a challenge, and the
server sends 18 bytes starting with 0x0D 0x0A and followed by 16 bytes
consisting of a challenge response. The details of this challenge and
other parts of the handshake are described in the next section.

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.
Clients and servers, after a successful handshake, transfer data back
and forth in conceptual units referred to in this specification as
"messages". A message is a complete unit of data at an application
level, with the expectation that many or most applications implementing
this protocol (such as web user agents) provide APIs in terms of
sending and receiving messages. On the network layer, a message may be
represented as one or more frames.
Data is sent on the wire in the form of frames that have an associated
type. Broadly speaking, there are types for textual data, which is
interpreted as UTF-8 text, binary data (whose interpretation is left up
to the application), and control frames, which are not intended to
carry data for the application, but instead for protocol-level
signalling, such as to signal that the connection should be closed.
The WebSocket protocol uses this framing so that specifications that
use the WebSocket 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.
To close the connection cleanly, a control frame is sent from one peer
to ask that the other peer close the connection. Details are specified
in Section 7 (Closing the connection).
The protocol is designed to support other frame types in future.
Currently only four frame types are defined -- continuation (used for
fragmented messages), control frames, text frames, and binary data
frames. Eight frame types are reserved for future use, and four frame
types are reserved for private use.
This wire format for the data transfer part is described by the ABNF
given in detail in Section 4 (Data Framing). A high level overview of
the framing is given in the following figure. [RFC5234] (Crocker, D.
and P. Overell, "Augmented BNF for Syntax Specifications: ABNF,"
January 2008.)

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-------+-+-------------+-------------------------------+
|M|R|R|R| opcode|R| Payload len |    Extended payload length    |
|O|S|S|S|  (4)  |S|     (7)     |             (16/63)           |
|R|V|V|V|       |V|             |   (if payload len==126/127)   |
|E|1|2|3|       |4|             |                               |
+-+-+-+-+-------+-+-------------+ - - - - - - - - - - - - - - - +
|     Extended payload length continued, if payload len == 127  |
+ - - - - - - - - - - - - - - - +-------------------------------+
|                               |          Extension data       |
+-------------------------------+ - - - - - - - - - - - - - - - +
:                                                               :
+---------------------------------------------------------------+
:                       Application data                        :
+---------------------------------------------------------------+
```
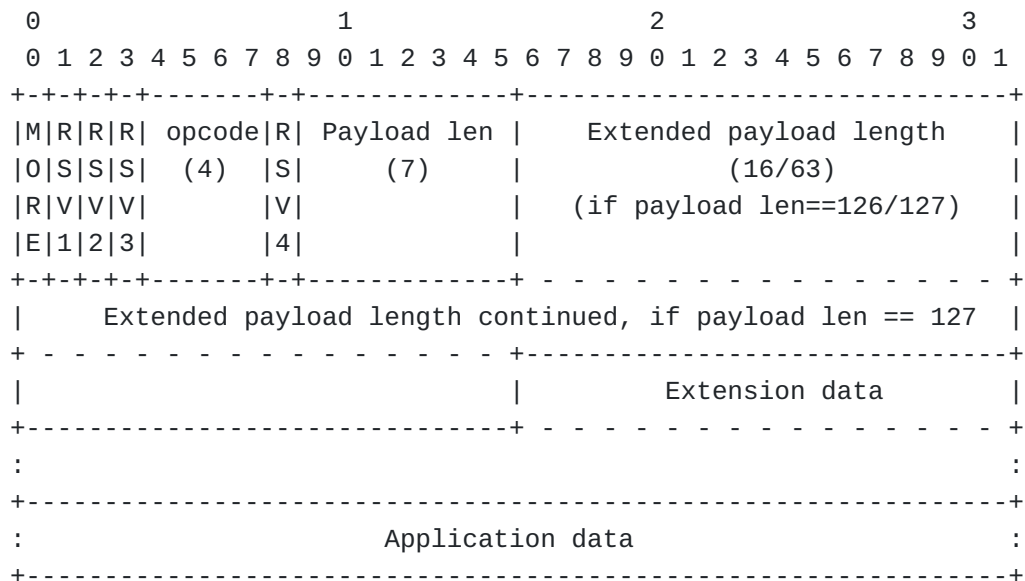
MORE: 1 bit
    Indicates more fragments follow in the current message

RSV1, RSV2, RSV3, RSV4: 1 bit each
    Must be 0 unless an extension is negotiated which defines
    meanings for non-zero values

Opcode: 4 bits
    Defines the interpretation of the payload data

Payload length: 7 bits
    The length of the payload: if 0-125, that is the payload length.
    If 126, the following 2 bytes interpreted as a 16 bit unsigned
    integer are the payload length. If 127, the following 8 bytes

interpreted as a 64-bit unsigned integer (the high bit must be 0)
are the payload length. The payload length is the length of the
Extension data + the length of the Application Data. The length
of the Extension data may be zero, in which case the Payload
length is the length of the Application data.

Extension data: n bytes
Only present if an extension is negotiated during the handshake
which defines it. If present, it is included in the total payload
length.

Application data: n bytes
Arbitrary application data, taking up the remainder of the frame
after any extension data. The length of the Application data is
equal to the payload length minus the length of the Extension
data.

1.3. Opening handshake

This section is non-normative.
The opening handshake is intended to be compatible with HTTP-based
server-side software, so that a single port can be used by both HTTP
clients talking to that server and WebSocket clients talking to that
server. To this end, the WebSocket client's handshake appears to HTTP
servers to be a regular GET request with an Upgrade offer:

```
GET / HTTP/1.1
Upgrade: WebSocket
Connection: Upgrade
```

Fields in the handshake are sent by the client in a random order; the
order is not meaningful.
Additional fields are used to select options in the WebSocket protocol.
The only options available in this version are the subprotocol
selector, |Sec-WebSocket-Protocol|, and |Cookie|, which can used for
sending cookies to the server (e.g. as an authentication mechanism).
The |Sec-WebSocket-Protocol| field takes an arbitrary string:

```
Sec-WebSocket-Protocol: chat
```

This field indicates the subprotocol (the application-level protocol
layered over the WebSocket protocol) that the client intends to use.
The server echoes this field in its handshake to indicate that it
supports that subprotocol.

The other fields in the handshake are all security-related. The |Host| field is used to protect against DNS rebinding attacks and to allow multiple domains to be served from one IP address.

        Host: example.com

The server includes the hostname in the |Sec-WebSocket-Location| field of its handshake, so that both the client and the server can verify that they agree on which host is in use.
The |Origin| field is used to protect against unauthorized cross-origin use of a WebSocket server by scripts using the |WebSocket| API in a Web browser. The server specifies which origin it is willing to receive requests from by including a |Sec-WebSocket-Origin| field with that origin. If multiple origins are authorized, the server echoes the value in the |Origin| field of the client's handshake.

        Origin: http://example.com

Finally, the server has to prove to the client that it received the client's WebSocket handshake, so that the server doesn't accept connections that are not WebSocket connections. This prevents an attacker from tricking a WebSocket server by sending it carefully-crafted packets using |XMLHttpRequest| or a |form| submission.
To prove that the handshake was received, the server has to take three pieces of information and combine them to form a response. The first two pieces of information come from the |Sec-WebSocket-Key1| and |Sec-WebSocket-Key2| fields in the client handshake:

        Sec-WebSocket-Key1: 18x 6]8vM;54 *(5: {   U1]8 z [8
        Sec-WebSocket-Key2: 1_ tx7X d < nw 334J702) 7]o}` 0

For each of these fields, the server has to take the digits from the value to obtain a number (in this case 1868545188 and 1733470270 respectively), then divide that number by the number of spaces characters in the value (in this case 12 and 10) to obtain a 32-bit number (155712099 and 173347027). These two resulting numbers are then used in the server handshake, as described below.
The counting of spaces is intended to make it impossible to smuggle this field into the resource name; making this even harder is the presence of *two* such fields, and the use of a newline as the only reliable indicator that the end of the key has been reached. The use of random characters interspersed with the spaces and the numbers ensures that the implementor actually looks for spaces and newlines, instead of being treating any character like a space, which would make it again easy to smuggle the fields into the path and trick the server. Finally, *dividing* by this number of spaces is intended to make sure that even the most naïve of implementations will check for spaces, since if ther

server does not verify that there are some spaces, the server will try to divide by zero, which is usually fatal (a correct handshake will always have at least one space).
The third piece of information is given after the fields, in the last eight bytes of the handshake, expressed here as they would be seen if interpreted as ASCII:

        Tm[K T2u

The concatenation of the number obtained from processing the |Sec-WebSocket-Key1| field, expressed as a big-endian 32 bit number, the number obtained from processing the |Sec-WebSocket-Key2| field, again expressed as a big-endian 32 bit number, and finally the eight bytes at the end of the handshake, form a 128 bit string whose MD5 sum is then used by the server to prove that it read the handshake.

The handshake from the server is much simpler than the client handshake. The first line is an HTTP Status-Line, with the status code 101 (the HTTP version and reason phrase aren't important):

        HTTP/1.1 101 WebSocket Protocol Handshake

The fields follow. Two of the fields are just for compatibility with HTTP:

        Upgrade: WebSocket
        Connection: Upgrade

Two of the fields are part of the security model described above, echoing the origin and stating the exact host, port, resource name, and whether the connection is expected to be encrypted:

        Sec-WebSocket-Origin: http://example.com
        Sec-WebSocket-Location: ws://example.com/

These fields are checked by the Web browser when it is acting as a | WebSocket| client for scripted pages. A server that only handles one origin and only serves one resource can therefore just return hard-coded values and does not need to parse the client's handshake to verify the correctness of the values.
Option fields can also be included. In this version of the protocol, the main option field is |Sec-WebSocket-Protocol|, which indicates the subprotocol that the server speaks. Web browsers verify that the server included the same value as was specified in the |WebSocket| constructor, so a server that speaks multiple subprotocols has to make

sure it selects one based on the client's handshake and specifies the right one in its handshake.

```
Sec-WebSocket-Protocol: chat
```

The server can also set cookie-related option fields to *set* cookies, as in HTTP.
After the fields, the server sends the aforementioned MD5 sum, a 16 byte (128 bit) value, shown here as if interpreted as ASCII:

```
fQJ,fN/4F4!~K~MH
```

This value depends on what the client sends, as described above. If it doesn't match what the client is expecting, the client would disconnect.
Having part of the handshake appear after the fields ensures that both the server and the client verify that the connection is not being interrupted by an HTTP intermediary such as a man-in-the-middle cache or proxy.

1.4. Closing handshake

This section is non-normative.
The closing handshake is far simpler than the opening handshake.
Either peer can send a control frame with data containing a specified control sequence to begin the closing handshake. Upon receiving such a frame, the other peer sends an identical frame in acknowledgement, if it hasn't already sent one. Upon receiving *that* control frame, the first peer then closes the connection, safe in the knowledge that no further data is forthcoming.
After sending a control frame indicating the connection should be closed, a peer does not send any further data; after receiving a control frame frame indicating the connection should be closed, a peer discards any further data received.
It is safe for both peers to initiate this handshake simultaneously.
The closing handshake is intended to replace the TCP closing handshake (FIN/ACK), on the basis that the TCP closing handshake is not always reliable end-to-end, especially in the presence of man-in-the-middle proxies and other intermediaries.

1.5. Design philosophy

This section is non-normative.
The WebSocket 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 WebSocket by the application
layer, in the same way that metadata is layered on top of TCP by the
application layer (HTTP).
Conceptually, WebSocket is really just a layer on top of TCP 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; 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; and reimplements the closing handshake
in-band. Other than that, it adds nothing. Basically it is intended to
be as close to just exposing raw TCP 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.
The protocol is intended to be extensible; future versions will likely
introduce a mechanism to compress data and might support sending binary
data.

1.6. Security model

This section is non-normative.
The WebSocket protocol uses the origin model used by Web browsers to
restrict which Web pages can contact a WebSocket server when the
WebSocket protocol is used from a Web page. Naturally, when the
WebSocket protocol is used by a dedicated client directly (i.e. not
from a Web page through a Web browser), 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).
It is similarly intended to fail to establish a connection when data
from other protocols, especially HTTP, is sent to a WebSocket server,
for example as might happen if an HTML |form| were submitted to a
WebSocket server. This is primarily achieved by requiring that the
server prove that it read the handshake, which it can only do if the
handshake contains the appropriate parts which themselves can only be

sent by a WebSocket handshake; in particular, fields starting with |
Sec-| cannot be set by an attacker from a Web browser, even when using
|XMLHttpRequest|.

1.7. Relationship to TCP and HTTP

This section is non-normative.
The WebSocket 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 WebSocket protocol
by default uses port 80 for regular WebSocket connections and port 443
for WebSocket connections tunneled over TLS.

1.8. Establishing a connection

This section is non-normative.
There are several options for establishing a WebSocket connection.
On the face of it, the simplest method would seem to be to use port 80
to get a direct connection to a WebSocket server. Port 80 traffic,
however, will often be intercepted by man-in-the-middle 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 WebSocket 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 WebSocket protocol to be deployed. In more elaborate setups (e.g.
with load balancers and multiple servers), a dedicated set of hosts for
WebSocket connections separate from the HTTP servers is probably easier
to manage.

1.9. Subprotocols using the WebSocket protocol

This section is non-normative.

The client can request that the server use a specific subprotocol by including the |Sec-Websocket-Protocol| field in its handshake. If it is specified, the server needs to include the same field 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 WebSocket 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 "v2.bookings.example.net". These subprotocols would be considered completely separate by WebSocket 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] (Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels," March 1997.)

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).
Comparing two strings in an **ASCII case-insensitive** manner means
comparing them exactly, code point for code point, except that the
characters in the range U+0041 to U+005A (i.e. LATIN CAPITAL LETTER A
to LATIN CAPITAL LETTER Z) and the corresponding characters in the
range U+0061 to U+007A (i.e. LATIN SMALL LETTER A to LATIN SMALL LETTER
Z) are considered to also match.
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. [HTML] (Hickson,
I., "HTML," August 2010.)
When an implementation is required to *send* data as part of the
WebSocket protocol, the implementation may delay the actual
transmission arbitrarily, e.g. buffering data so as to send fewer IP
packets.

---

3. WebSocket URLs

3.1. Parsing WebSocket URLs

The steps to **parse a WebSocket 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] (Connolly, D. and C. Sperberg-
   McQueen, "Web addresses in HTML 5," May 2009.)

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] (Connolly, D. and C. Sperberg-McQueen, "Web addresses in HTML 5," May 2009.) [RFC3629] (Yergeau, F., "UTF-8, a transformation format of ISO 10646," November 2003.)

   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 WebSocket URLs

The steps to **construct a WebSocket 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/.

3.3. Valid WebSocket URLs

For a WebSocket URL to be considered valid, the following conditions MUST hold.

*The /host/ must be ASCII-only (i.e. it must have been punycode-encoded already if necessary).

*The /origin/ must not contain characters in the range U+0041 to U+005A (i.e. LATIN CAPITAL LETTER A to LATIN CAPITAL LETTER Z).

*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.

Any WebSocket URLs not meeting the above criteria are considered invalid, and a client MUST NOT attempt to make a connection to an invalid WebSocket URL. A client SHOULD attempt to parse a URL obtained from any external source (such as a web site or a user) using the steps specified in Section 3.1 (Parsing WebSocket URLs) to obtain a valid WebSocket URL, but MUST NOT attempt to connect with such an unparsed URL, and instead only use the parsed version and only if that version is considered valid by the criteria above.

---

4. Data Framing

4.1. Overview

The base framing protocol is deliberately kept simple so that simple implementations may ignore advanced features. In the absence of extensions negotiated during the opening handshake (Section 5 (Opening Handshake)), all reserved bits must be 0 and no reserved opcode values may be used.

4.2. Base Framing Protocol

The base framing protocol is defined by the following ABNF [RFC5234] (Crocker, D. and P. Overell, "Augmented BNF for Syntax Specifications: ABNF," January 2008.):

```
ws-frame               = frame-more
                         frame-rsv1
                         frame-rsv2
                         frame-rsv3
                         frame-opcode
                         frame-rsv4
                         frame-length
                         frame-extension
                         application-data;

frame-more             = %x0 ; final frame of message
                       / %x1 ; more frames of this message follow

frame-rsv1             = %x0 ; 1 bit, must be 0

frame-rsv2             = %x0 ; 1 bit, must be 0

frame-rsv3             = %x0 ; 1 bit, must be 0

frame-opcode           = %x0 ; continuation frame
                       / %x1 ; connection close
                       / %x2 ; ping
                       / %x3 ; pong
                       / %x4 ; text frame
                       / %x5 ; binary frame
                       / %x6-F ; reserved

frame-rsv4             = %x0 ; 1 bit, must be 0

frame-length           = %x00-7D
                       / %x7E frame-length-16
                       / %x7F frame-length-63

frame-length-16        = %x0000-FFFF

frame-length-63        = %x0000000000000000-7FFFFFFFFFFFFFFF

frame-extension        = *( %x00-FF ) ; to be defined later

application-data       = *( %x00-FF )
```

4.3. Fragmentation

The following rules apply to fragmentation:

* *An unfragmented message consists of a single frame with the MORE bit clear and an opcode other than 0.

* *A fragmented message consists of a single frame with the MORE bit set and an opcode other than 0, followed by zero or more frames with the MORE bit set and the opcode set to 0, and terminated by a single frame with the MORE bit clear and an opcode of 0. Its content is the concatenation of the application data from each of those frames in order.

* *Note: There is an open question as to whether control frames be interjected in the middle of a fragmented message. If so, it must be decided whether they be fragmented (which would require keeping a stack of "in-progress" messages).*

* *A sender MAY arbitrarily fragment a single message (which allows generation of dynamic content without having to buffer the data in order to count it).

* *A receiver MUST be prepared to accept arbitrarily fragmented messages, even if the sender sent the message in a single frame.

* *An intermediary MAY fragment a message arbitrarily, except that it MUST NOT fragment or otherwise modify any message with any reserved bits set or using any reserved opcode, unless it observed the negotiation of an extension which it understands and which defines the interpretation of those values.

4.4. Control Frames

The Close (0x01), Ping (0x02), and Pong (0x03) frames are contol frames -- they do not supply data to the ultimate endpoint, but instead are used to carry out tasks related to the WebSocket connection itself. A receiver MUST take the following action upon receiving control frames:

Close:
> Upon receipt of a close frame, an endpoint SHOULD send a Close frame to the remote recipient, if it has not already done so, deliver a close event to the application if necessary, and then close the WebSocket.

Ping

Upon receipt of a Ping message, an endpoint SHOULD send a Pong response as soon as is practical. The Pong response MUST contain the payload provided in the Ping message, though an implementation MAY truncate the message at an implementation-defined size which MUST be at least 8 *(TBD)* bytes.

Ping frames MAY be sent as a keep-alive mechanism, but if so the interval SHOULD be configurable.

Pong

If a Pong message is received without a matching Ping message being sent, an endpoint MUST drop the connection. Otherwise, the endpoint SHOULD update any liveness timer it may have for the connection.

4.5. Data Frames

All frame types not listed above are data frames, which transport application-layer data. The opcode determines the interpretation of the application data:

Text

The payload data is text data encoded as UTF-8.

Binary

The payload data is arbitrary binary data whose interpretation is solely up to the application layer.

Additional data frame types will be defined in extensions or in a subsequent version of the protocol.

4.6. Examples

This section is non-normative.

*A single-frame text message

-0x04 0x05 "Hello"

*A fragmented text message

-0x84 0x03 "Hel"

-0x00 0x02 "lo"

*Ping request and response

-0x02 0x05 "Hello"

-0x03 0x05 "Hello"

*256 bytes binary message in a single frame

-0x05 0x7E 0x0100 [256 bytes of binary data]

*64KiB binary message in a single frame

-0x05 0x7F 0x0000000000010000 [65536 bytes of binary data]

4.7. Extensibility

Extensions will be defined which extend the base protocol, but only if
their use is negotiated during the handshake. The following mechanisms
will be used for extension:

*Extension data may be placed in the payload before the
 application data.

*Reserved bits can be allocated for per-frame needs.

*Reserved opcode values can be defined.

*Reserved bits can be allocated to the opcode field if more opcode
 values are needed.

*A reserved bit or an "extension" opcode can be defined which
 allocates additional bits out of the payload area to define
 larger opcodes or more per-frame bits.

---

5. Opening Handshake

5.1. Client Requirements

When the user agent is to **establish a WebSocket connection** to a host /
host/, on a port /port/, from an origin whose ASCII serialization is /
origin/, with a flag /secure/, with a string giving a /resource name/,
and optionally with a string giving a /protocol/, it must run the
following steps. [ORIGIN] (Barth, A., Jackson, C., and I. Hickson, "The
HTTP Origin Header," September 2009.)

1. Verify that the WebSocket URL and its components are valid
    according to Section 3.3 (Valid WebSocket URLs). If any of the
    requirements are not met, the client MUST fail the WebSocket
    connection and abort these steps.

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

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

    NOTE: There is no limit to the number of established WebSocket
    connections a user agent can have with a single remote host.
    Servers can refuse to connect users with an excessive number of
    connections, or disconnect resource-hogging users when
    suffering high load.

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

    > EXAMPLE: For example, if the user agent uses an HTTP proxy
    > for all traffic, then if it was to try to connect to port 80
    > on server example.com, it might send the following lines to
    > the proxy server:
    >
    >
    >   CONNECT example.com:80 HTTP/1.1
    >   Host: example.com
    >
    >
    > If there was a password, the connection might look like:

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

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

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

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

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

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

5. If /secure/ is true, perform a TLS handshake over the connection. If this fails (e.g. the server's certificate could not be verified), then fail the WebSocket connection and abort these steps. Otherwise, all further communication on this channel must run through the encrypted tunnel. [RFC2246] (Dierks, T. and C. Allen, "The TLS Protocol Version 1.0," January 1999.)

   User agents must use the Server Name Indication extension in the TLS handshake. [RFC4366] (Blake-Wilson, S., Nystrom, M., Hopwood, D., Mikkelsen, J., and T. Wright, "Transport Layer Security (TLS) Extensions," April 2006.)

6. Send the UTF-8 string "GET" followed by a UTF-8-encoded U+0020 SPACE character to the remote side (the server).

   Send the /resource name/ value, encoded as UTF-8.

   Send another UTF-8-encoded U+0020 SPACE character, followed by the UTF-8 string "HTTP/1.1", followed by a UTF-8-encoded U+000D CARRIAGE RETURN U+000A LINE FEED character pair (CRLF).

7. Let /fields/ be an empty list of strings.

8. Add the string "Upgrade: WebSocket" to /fields/.

9. Add the string "Connection: Upgrade" to /fields/.

10. Let /hostport/ be an empty string.

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

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

13. Add the string consisting of the concatenation of the string "Host:", a U+0020 SPACE character, and /hostport/, to /fields/.

14. Add the string consisting of the concatenation of the string "Origin:", a U+0020 SPACE character, and the /origin/ value, to /fields/.

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

    Otherwise, add the string consisting of the concatenation of the string "Sec-WebSocket-Protocol:", a U+0020 SPACE character, and the /protocol/ value, to /fields/.

16. 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 add to /fields/ any HTTP headers that would be appropriate for that information. [RFC2616] (Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., and T. Berners-Lee, "Hypertext Transfer Protocol -- HTTP/1.1," June 1999.) [RFC2109] (Kristol, D. and L. Montulli, "HTTP State Management Mechanism," February 1997.) [RFC2965] (Kristol, D. and L. Montulli, "HTTP State Management Mechanism," October 2000.)

    This includes "HttpOnly" cookies (cookies with the http-only-flag set to true); the WebSocket protocol is not considered a non-HTTP API for the purpose of cookie processing.

17. Let /spaces_1/ be a random integer from 1 to 12 inclusive.

    Let /spaces_2/ be a random integer from 1 to 12 inclusive.

    EXAMPLE: For example, 5 and 9.

18. Let /max_1/ be the largest integer not greater than 4,294,967,295 divided by /spaces_1/.

    Let /max_2/ be the largest integer not greater than 4,294,967,295 divided by /spaces_2/.

    EXAMPLE: Continuing the example, 858,993,459 and 477,218,588.

19. Let /number_1/ be a random integer from 0 to /max_1/ inclusive.

    Let /number_2/ be a random integer from 0 to /max_2/ inclusive.

    EXAMPLE: For example, 777,007,543 and 114,997,259.

20. Let /product_1/ be the result of multiplying /number_1/ and /spaces_1/ together.

    Let /product_2/ be the result of multiplying /number_2/ and /spaces_2/ together.

    EXAMPLE: Continuing the example, 3,885,037,715 and 1,034,975,331.

21. Let /key_1/ be a string consisting of /product_1/, expressed in base ten using the numerals in the range U+0030 DIGIT ZERO (0) to U+0039 DIGIT NINE (9).

    Let /key_2/ be a string consisting of /product_2/, expressed in base ten using the numerals in the range U+0030 DIGIT ZERO (0) to U+0039 DIGIT NINE (9).

    EXAMPLE: Continuing the example, "3885037715" and "1034975331".

22. Insert between one and twelve random characters from the ranges U+0021 to U+002F and U+003A to U+007E into /key_1/ at random positions.

    Insert between one and twelve random characters from the ranges U+0021 to U+002F and U+003A to U+007E into /key_2/ at random positions.

    NOTE: This corresponds to random printable ASCII characters other than the digits and the U+0020 SPACE character.

    EXAMPLE: Continuing the example, this could lead to "P388O503D&ul7{K%gX(%715" and "1N?|kUT0or3o4I97N5-S3O31".

23. Insert /spaces_1/ U+0020 SPACE characters into /key_1/ at
    random positions other than the start or end of the string.

    Insert /spaces_2/ U+0020 SPACE characters into /key_2/ at
    random positions other than the start or end of the string.

    EXAMPLE: Continuing the example, this could lead to
    "P388 O503D&ul7 {K%gX(%7 15" and "1 N ?|k UT0or 3o 4 I97N 5-
    S3O 31".

24. Add the string consisting of the concatenation of the string
    "Sec-WebSocket-Key1:", a U+0020 SPACE character, and the /
    key_1/ value, to /fields/.

    Add the string consisting of the concatenation of the string
    "Sec-WebSocket-Key2:", a U+0020 SPACE character, and the /
    key_2/ value, to /fields/.

25. For each string in /fields/, in a random order: send the
    string, encoded as UTF-8, followed by a UTF-8-encoded U+000D
    CARRIAGE RETURN U+000A LINE FEED character pair (CRLF). It is
    important that the fields be output in a random order so that
    servers not depend on the particular order used by any
    particular client.

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

27. Let /key3/ be a string consisting of eight random bytes (or
    equivalently, a random 64 bit integer encoded in big-endian
    order).

    EXAMPLE: For example, 0x47 0x30 0x22 0x2D 0x5A 0x3F 0x47 0x58.

28. Send /key3/ to the server.

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

    If /field/ is not at least seven bytes long, or if the last two
    bytes aren't 0x0D and 0x0A respectively, or if it does not
    contain at least two 0x20 bytes, then fail the WebSocket
    connection and abort these steps.

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

30. Let /code/ be the substring of /field/ that starts from the byte after the first 0x20 byte, and ends with the byte before the second 0x20 byte.

31. If /code/ is not three bytes long, or if any of the bytes in /code/ are not in the range 0x30 to 0x39, then fail the WebSocket connection and abort these steps.

32. If /code/, interpreted as UTF-8, is "101", then move to the next step.

    If /code/, interpreted as UTF-8, is "407", then either close the connection and jump back to step 2, providing appropriate authentication information, or fail the WebSocket 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).

    Otherwise, fail the WebSocket connection and abort these steps.

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

34. *Field*: Let /name/ and /value/ be empty byte arrays.

35. Read a byte from the server.

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

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

    > **-> If the byte is 0x0D (ASCII CR)** If the /name/ byte array is empty, then jump to the fields processing step. Otherwise, fail the WebSocket connection and abort these steps.
    >
    > **-> If the byte is 0x0A (ASCII LF)** Fail the WebSocket 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 field name, terminated by a colon, converting upper-case ASCII letters to lowercase, and aborting if a stray CR or LF is found.

36. Let /count/ equal 0.

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

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

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

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

    > **-> If the byte is 0x20 (ASCII space) and /count/ equals 1**
    >   Ignore the byte and redo this step for the next byte.
    >
    > **-> If the byte is 0x0D (ASCII CR)** Move on to the next step.
    >
    > **-> If the byte is 0x0A (ASCII LF)** Fail the WebSocket
    >   connection and abort these steps.
    >
    > **-> Otherwise** Append the byte to the /value/ byte array and
    >   redo this step for the next byte.

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

38. 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 WebSocket connection and abort these steps.

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

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

40. Return to the "Field" step above.

41. *Fields processing*: Read a byte from the server.

    If the connection closes before this byte is received, or if the byte is not a 0x0A byte (ASCII LF), then fail the WebSocket

connection and abort these steps.

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

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

> **-> If the entry's name is "upgrade"** If the value is not
> exactly equal to the string "WebSocket", then fail the
> WebSocket connection and abort these steps.

> **-> If the entry's name is "connection"** If the value,
> converted to ASCII lowercase, is not exactly equal to the
> string "upgrade", then fail the WebSocket connection and
> abort these steps.

> **-> If the entry's name is "sec-websocket-origin"** If the
> value is not exactly equal to /origin/, then fail the
> WebSocket connection and abort these steps. [ORIGIN]
> (Barth, A., Jackson, C., and I. Hickson, "The HTTP Origin
> Header," September 2009.)

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

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

> **-> If the entry's name is "set-cookie" or "set-cookie2" or
> another cookie-related field name**
> If the relevant specification is supported by the user
> agent, 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] (Kristol, D. and L. Montulli, "HTTP State
Management Mechanism," February 1997.) [RFC2965]
(Kristol, D. and L. Montulli, "HTTP State Management
Mechanism," October 2000.)

If the relevant specification is not supported by the
user agent, then the field must be ignored.

-> Any other name Ignore it.

43. Let /challenge/ be the concatenation of /number_1/, expressed
    as a big-endian 32 bit integer, /number_2/, expressed as a big-
    endian 32 bit integer, and the eight bytes of /key_3/ in the
    order they were sent on the wire.

    EXAMPLE: Using the examples given earlier, this leads to the 16
    bytes 0x2E 0x50 0x31 0xB7 0x06 0xDA 0xB8 0x0B 0x47 0x30 0x22
    0x2D 0x5A 0x3F 0x47 0x58.

44. Let /expected/ be the MD5 fingerprint of /challenge/ as a big-
    endian 128 bit string. [RFC1321] (Rivest, R., "The MD5 Message-
    Digest Algorithm," April 1992.)

    EXAMPLE: Using the examples given earlier, this leads to the 16
    bytes 0x30 0x73 0x74 0x33 0x52 0x6C 0x26 0x71 0x2D 0x32 0x5A
    0x55 0x5E 0x77 0x65 0x75. In ASCII, these bytes correspond to
    the string "0st3Rl&q-2ZU^weu".

45. Read sixteen bytes from the server. Let /reply/ be those bytes.

    If the connection closes before these bytes are received, then
    fail the WebSocket connection and abort these steps.

46. If /reply/ does not exactly equal /expected/, then fail the
    WebSocket connection and abort these steps.

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

5.2. Server-side requirements

This section only applies to servers.

5.2.1. Reading the client's opening handshake

When a client starts a WebSocket connection, it sends its part of the opening handshake. The server must parse at least part of this handshake in order to obtain the necessary information to generate the server part of the handshake.
The client handshake consists of the following parts. If the server, while reading the handshake, finds that the client did not send a handshake that matches the description below, the server should abort the WebSocket connection.

1. The three-character UTF-8 string "GET".

2. A UTF-8-encoded U+0020 SPACE character (0x20 byte).

3. A string consisting of all the bytes up to the next UTF-8-encoded U+0020 SPACE character (0x20 byte). The result of decoding this string as a UTF-8 string is the name of the resource requested by the server. If the server only supports one resource, then this can safely be ignored; the client verifies that the right resource is supported based on the information included in the server's own handshake. The resource name will begin with U+002F SOLIDUS character (/) and will only include characters in the range U+0021 to U+007E.

4. A string of bytes terminated by a UTF-8-encoded U+000D CARRIAGE RETURN U+000A LINE FEED character pair (CRLF). All the characters from the second 0x20 byte up to the first 0x0D 0x0A byte pair in the data from the client can be safely ignored. (It will probably be the string "HTTP/1.1".)

5. A series of fields.

   Each field is terminated by a UTF-8-encoded U+000D CARRIAGE RETURN U+000A LINE FEED character pair (CRLF). The end of the fields is denoted by the terminating CRLF pair being followed immediately by another CRLF pair.

   NOTE: In other words, the fields start with the first 0x0D 0x0A byte pair, end with the first 0x0D 0x0A 0x0D 0x0A byte sequence, and are separate from each other by 0x0D 0x0A byte pairs.

The fields are encoded as UTF-8.

Each field consists of a name, consisting of one or more
characters in the ranges U+0021 to U+0039 and U+003B to U+007E,
followed by a U+003A COLON character (:) and a U+0020 SPACE
character, followed by zero or more characters forming the
value.

The expected field names, the meaning of their corresponding
values, and the processing servers are required to apply to
those fields, are described below, after the description of the
client handshake.

6. After the first 0x0D 0x0A 0x0D 0x0A byte sequence, indicating
   the end of the fields, the client sends eight random bytes.
   These are used in constructing the server handshake.

The expected field names, and the meaning of their corresponding
values, are as follows. Field names must be compared in an ASCII case-
insensitive manner.

|Upgrade| Invariant part of the handshake. Will always have a value
    that is an ASCII case-insensitive match for the string
    "WebSocket".

    Can be safely ignored, though the server should abort the
    WebSocket connection if this field is absent or has a different
    value, to avoid vulnerability to cross-protocol attacks.

|Connection| Invariant part of the handshake. Will always have a
    value that is an ASCII case-insensitive match for the string
    "Upgrade".

    Can be safely ignored, though the server should abort the
    WebSocket connection if this field is absent or has a different
    value, to avoid vulnerability to cross-protocol attacks.

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

    Can be safely ignored, though the server should abort the
    WebSocket connection if this field is absent or has a value that

does not match the server's host name, to avoid vulnerability to cross-protocol attacks and DNS rebinding attacks.

|**Origin**| The value gives the scheme, hostname, and port (if it's not the default port for the given scheme) of the page that asked the client to open the WebSocket. It would be interesting if the server's operator had deals with operators of other sites, since the server could then decide how to respond (or indeed, *whether* to respond) based on which site was requesting a connection. [ORIGIN] (Barth, A., Jackson, C., and I. Hickson, "The HTTP Origin Header," September 2009.)

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

|**Sec-WebSocket-Protocol**| 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.

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

|**Sec-WebSocket-Key1**| The values provide the information required
|**Sec-WebSocket-Key2**| for computing the server's handshake, as described in the next section.

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.

Unrecognized fields can be safely ignored, and are probably either the result of intermediaries injecting fields unrelated to the operation of the WebSocket protocol, or clients that support future versions of the protocol offering options that the server doesn't support.

5.2.2. Sending the server's opening handshake

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

1. If the server supports encryption, perform a TLS handshake over
   the connection. If this fails (e.g. the client indicated a host
   name in the extended client hello "server_name" extension that
   the server does not host), then close the connection;
   otherwise, all further communication for the connection
   (including the server handshake) must run through the encrypted
   tunnel. [RFC2246] (Dierks, T. and C. Allen, "The TLS Protocol
   Version 1.0," January 1999.)

2. Establish the following information:

   /host/ The host name or IP address of the WebSocket server,
      as it is to be addressed by clients. The host name must
      be punycode-encoded if necessary. If the server can
      respond to requests to multiple hosts (e.g. in a virtual
      hosting environment), then the value should be derived
      from the client's handshake, specifically from the "Host"
      field.

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

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

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

   /origin/ The ASCII serialization of the origin that the
      server is willing to communicate with, converted to ASCII
      lowercase. If the server can respond to requests from
      multiple origins (or indeed, all origins), then the value
      should be derived from the client's handshake,
      specifically from the "Origin" field. [ORIGIN] (Barth,
      A., Jackson, C., and I. Hickson, "The HTTP Origin
      Header," September 2009.)

   /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 "Sec-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.

/key_1/ The value of the "Sec-WebSocket-Key1" field in the
client's handshake.

/key_2/ The value of the "Sec-WebSocket-Key2" field in the
client's handshake.

/key_3/ The eight random bytes sent after the first 0x0D
0x0A 0x0D 0x0A sequence in the client's handshake.

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

4. Let /key-number_1/ be the digits (characters in the range
U+0030 DIGIT ZERO (0) to U+0039 DIGIT NINE (9)) in /key_1/,
interpreted as a base ten integer, ignoring all other
characters in /key_1/.

Let /key-number_2/ be the digits (characters in the range
U+0030 DIGIT ZERO (0) to U+0039 DIGIT NINE (9)) in /key_2/,
interpreted as a base ten integer, ignoring all other
characters in /key_2/.

EXAMPLE: For example, assume that the client handshake was:

```
GET / HTTP/1.1
Connection: Upgrade
Host: example.com
Upgrade: WebSocket
Sec-WebSocket-Key1: 3e6b263  4 17 80
Origin: http://example.com
Sec-WebSocket-Key2: 17  9 G`ZD9   2 2b 7X 3 /r90

WjN}|M(6
```

The /key-number_1/ would be the number 3,626,341,780, and
the /key-number_2/ would be the number 1,799,227,390.

In this example, incidentally, /key_3/ is "WjN}|M(6", or
0x57 0x6A 0x4E 0x7D 0x7C 0x4D 0x28 0x36.

5. Let /spaces_1/ be the number of U+0020 SPACE characters in /
key_1/.

Let /spaces_2/ be the number of U+0020 SPACE characters in /

key_2/.

If either /spaces_1/ or /spaces_2/ is zero, then abort the
WebSocket connection. This is a symptom of a cross-protocol
attack.

EXAMPLE: In the example above, /spaces_1/ would be 4 and /
spaces_2/ would be 10.

6. If /key-number_1/ is not an integral multiple of /spaces_1/,
   then abort the WebSocket connection.

   If /key-number_2/ is not an integral multiple of /spaces_2/,
   then abort the WebSocket connection.

   NOTE: This can only happen if the client is not a conforming
   WebSocket client.

7. Let /part_1/ be /key-number_1/ divided by /spaces_1/.

   Let /part_2/ be /key-number_2/ divided by /spaces_2/.

   EXAMPLE: In the example above, /part_1/ would be 906,585,445
   and /part_2/ would be 179,922,739.

8. Let /challenge/ be the concatenation of /part_1/, expressed as
   a big-endian 32 bit integer, /part_2/, expressed as a big-
   endian 32 bit integer, and the eight bytes of /key_3/ in the
   order they were sent on the wire.

   EXAMPLE: In the example above, this would be the 16 bytes 0x36
   0x09 0x65 0x65 0x0A 0xB9 0x67 0x33 0x57 0x6A 0x4E 0x7D 0x7C
   0x4D 0x28 0x36.

9. Let /response/ be the MD5 fingerprint of /challenge/ as a big-
   endian 128 bit string. [RFC1321] (Rivest, R., "The MD5 Message-
   Digest Algorithm," April 1992.)

   EXAMPLE: In the example above, this would be the 16 bytes 0x6E
   0x60 0x39 0x65 0x42 0x6B 0x39 0x7A 0x24 0x52 0x38 0x70 0x4F
   0x74 0x56 0x62, or "n`9eBk9z$R8pOtVb" in ASCII.

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

                    HTTP/1.1 101 WebSocket Protocol Handshake

This line may be sent differently if necessary, but must match the Status-Line production defined in the HTTP specification, with the Status-Code having the value 101.

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

> **|Upgrade|** The value must be the string "WebSocket".
>
> **|Connection|** The value must be the string "Upgrade".
>
> **|Sec-WebSocket-Location|** The value must be /location/
>
> **|Sec-WebSocket-Origin|** The value must be /origin/
>
> **|Sec-WebSocket-Protocol|** This field must be included if / subprotocol/ is not null, and must not be included if / subprotocol/ is null.
>
> If included, the value must be /subprotocol/

Optionally, include "Set-Cookie", "Set-Cookie2", or other cookie-related fields, with values equal to the values that would be used for the identically named HTTP headers. [RFC2109] (Kristol, D. and L. Montulli, "HTTP State Management Mechanism," February 1997.) [RFC2965] (Kristol, D. and L. Montulli, "HTTP State Management Mechanism," October 2000.)

12. Send two bytes 0x0D 0x0A (ASCII CRLF).

13. Send /response/.

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

6. Error Handling

6.1. Handling errors in UTF-8 from the server

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.

6.2. Handling errors in UTF-8 from the client

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, behavior 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 WebSocket
protocol might define specific behavior for servers.

---

7. Closing the connection

7.1. Client-initiated closure

Certain algorithms require the user agent to **fail the WebSocket
connection**. To do so, the user agent must close the WebSocket
connection, and may report the problem to the user (which would be
especially useful for developers).
Except as indicated above or as specified by the application layer
(e.g. a script using the WebSocket API), user agents should not close
the connection.
User agents must not convey any failure information to scripts in a way
that would allow a script to distinguish the following situations:

   *A server whose host name could not be resolved.

   *A server to which packets could not successfully be routed.

   *A server that refused the connection on the specified port.

*A server that did not complete the opening handshake (e.g.
 because it was not a WebSocket server).

*A WebSocket server that sent a correct opening handshake, but
 that specified options that caused the client to drop the
 connection (e.g. the server specified an origin that differed
 from the script's).

*A WebSocket server that abruptly closed the connection after
 successfully completing the opening handshake.

7.2. Server-initiated closure

Certain algorithms require or recommend that the server **abort the
WebSocket connection** during the opening handshake. To do so, the server
must simply close the WebSocket connection.

7.3. Closure

To **close the WebSocket connection**, the user agent or server must close
the TCP connection, using whatever mechanism possible (e.g. either the
TCP RST or FIN mechanisms). 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 **WebSocket connection is closed**.
If the connection was closed after the client finished the WebSocket
closing handshake, then the WebSocket connection is said to have been
closed *cleanly*.
Servers may close the WebSocket connection whenever desired. User
agents should not close the WebSocket connection arbitrarily.

---

8. 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" fields, 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" field is an origin they expect, and should only respond with the corresponding "Sec-WebSocket-Origin" if it is an accepted origin. Servers that only accept input from one origin can just send back that value in the "Sec-WebSocket-Origin" field, 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.

---

9. IANA considerations

9.1. Registration of ws: scheme

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

URI scheme name. ws

Status. Permanent.

URI scheme syntax. In ABNF terms using the terminals from the URI specifications: [RFC5234] (Crocker, D. and P. Overell, "Augmented BNF for Syntax Specifications: ABNF," January 2008.) [RFC3986] (Berners-Lee, T., Fielding, R., and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax," January 2005.)

```
                "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 WebSocket 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] (Faltstrom, P.,
Hoffman, P., and A. Costello, "Internationalizing Domain Names in
Applications (IDNA)," March 2003.)

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] (Berners-Lee, T.,
Fielding, R., and L. Masinter, "Uniform Resource Identifier
(URI): Generic Syntax," January 2005.) [RFC3987] (Duerst, M. and
M. Suignard, "Internationalized Resource Identifiers (IRIs),"
January 2005.)

Applications/protocols that use this URI scheme name. WebSocket
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.

9.2. Registration of wss: scheme

A |wss:| URL identifies a WebSocket 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] (Crocker, D. and P. Overell, "Augmented
    BNF for Syntax Specifications: ABNF," January 2008.) [RFC3986]
    (Berners-Lee, T., Fielding, R., and L. Masinter, "Uniform
    Resource Identifier (URI): Generic Syntax," January 2005.)

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

    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] (Berners-Lee, T.,
    Fielding, R., and L. Masinter, "Uniform Resource Identifier
    (URI): Generic Syntax," January 2005.) [RFC3987] (Duerst, M. and
    M. Suignard, "Internationalized Resource Identifiers (IRIs),"
    January 2005.)

Applications/protocols that use this URI scheme name. WebSocket
    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.

9.3. 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.

9.4. Sec-WebSocket-Key1 and Sec-WebSocket-Key2

This section describes two header fields for registration in the
Permanent Message Header Field Registry. [RFC3864] (Klyne, G.,
Nottingham, M., and J. Mogul, "Registration Procedures for Message
Header Fields," September 2004.)

Header field name Sec-WebSocket-Key1

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.

Header field name Sec-WebSocket-Key2

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.

The |Sec-WebSocket-Key1| and |Sec-WebSocket-Key2| headers are used in
the WebSocket handshake. They are sent from the client to the server to
provide part of the information used by the server to prove that it
received a valid WebSocket handshake. This helps ensure that the server
does not accept connections from non-Web-Socket clients (e.g. HTTP
clients) that are being abused to send data to unsuspecting WebSocket
servers.

9.5. Sec-WebSocket-Location

This section describes a header field for registration in the Permanent
Message Header Field Registry. [RFC3864] (Klyne, G., Nottingham, M.,
and J. Mogul, "Registration Procedures for Message Header Fields,"
September 2004.)

Header field name   Sec-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.

The |Sec-WebSocket-Location| header is used in the WebSocket 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.

9.6. Sec-WebSocket-Origin

This section describes a header field for registration in the Permanent
Message Header Field Registry. [RFC3864] (Klyne, G., Nottingham, M.,
and J. Mogul, "Registration Procedures for Message Header Fields,"
September 2004.)

> **Header field name** Sec-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.

The |Sec-WebSocket-Origin| header is used in the WebSocket 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.

9.7. Sec-WebSocket-Protocol

This section describes a header field for registration in the Permanent
Message Header Field Registry. [RFC3864] (Klyne, G., Nottingham, M.,
and J. Mogul, "Registration Procedures for Message Header Fields,"
September 2004.)

> **Header field name** Sec-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.

The |Sec-WebSocket-Protocol| header is used in the WebSocket 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.

---

10. Using the WebSocket protocol from other specifications

The WebSocket 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 WebSocket 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] (Barth, A., Jackson, C.,
    and I. Hickson, "The HTTP Origin Header," September 2009.)

   *Optionally a string identifying a protocol that is to be layered
    over the WebSocket connection.

The /host/, /port/, /resource name/, and /secure/ flag are usually
obtained from a URL using the steps to parse a WebSocket URL's
components. These steps fail if the URL does not specify a WebSocket.
If a connection can be established, then it is said that the "WebSocket
connection is established".
If at any time the connection is to be closed, then the specification
needs to use the "close the WebSocket 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
"WebSocket connection is closed".
While a connection is open, the specification will need to handle the
cases when "a WebSocket message has been received" with text /data/.
To send some text /data/ to an open connection, the specification needs
to "send /data/ using the WebSocket".

---

11. Acknowledgements

Special thanks are due to Ian Hickson, who was the original author and editor of this protocol. The initial design of this specification benefitted from the participation of many people in the WHATWG and WHATWG mailing list. Contributions to that specification are not tracked by section, but a list of all who contributed to that specification is given in the WHATWG HTML specification. [HTML] (Hickson, I., "HTML," August 2010.)
Special thanks also to John Tamplin for providing a significant amount of text for the Data Framing section of this specification.

---

12. Normative References

[HTML]	Hickson, I., "HTML," August 2010.
[ORIGIN]	Barth, A., Jackson, C., and I. Hickson, "The HTTP Origin Header," draft-abarth-origin (work in progress), September 2009.
[RFC1321]	Rivest, R., "The MD5 Message-Digest Algorithm," RFC 1321, April 1992 (TXT).
[RFC2109]	Kristol, D. and L. Montulli, "HTTP State Management Mechanism," RFC 2109, February 1997 (TXT, HTML, XML).
[RFC2119]	Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels," BCP 14, RFC 2119, March 1997 (TXT, HTML, XML).
[RFC2246]	Dierks, T. and C. Allen, "The TLS Protocol Version 1.0," RFC 2246, January 1999 (TXT).
[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 (TXT, PS, PDF, HTML, XML).
[RFC2965]	Kristol, D. and L. Montulli, "HTTP State Management Mechanism," RFC 2965, October 2000 (TXT, HTML, XML).
[RFC3490]	Faltstrom, P., Hoffman, P., and A. Costello, "Internationalizing Domain Names in Applications (IDNA)," RFC 3490, March 2003 (TXT).
[RFC3629]	Yergeau, F., "UTF-8, a transformation format of ISO 10646," STD 63, RFC 3629, November 2003 (TXT).
[RFC3864]	Klyne, G., Nottingham, M., and J. Mogul, "Registration Procedures for Message Header Fields," BCP 90, RFC 3864, September 2004 (TXT).
[RFC3986]	Berners-Lee, T., Fielding, R., and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax," STD 66, RFC 3986, January 2005 (TXT, HTML, XML).
[RFC3987]	

	Duerst, M. and M. Suignard, "[Internationalized Resource Identifiers (IRIs)](#)," RFC 3987, January 2005 ([TXT](#)).
[RFC4366]	Blake-Wilson, S., Nystrom, M., Hopwood, D., Mikkelsen, J., and T. Wright, "[Transport Layer Security (TLS) Extensions](#)," RFC 4366, April 2006 ([TXT](#)).
[RFC5234]	Crocker, D. and P. Overell, "[Augmented BNF for Syntax Specifications: ABNF](#)," STD 68, RFC 5234, January 2008 ([TXT](#)).
[WEBADDRESSES]	Connolly, D. and C. Sperberg-McQueen, "[Web addresses in HTML 5](#)," May 2009.
[WSAPI]	Hickson, I., "[The Web Sockets API](#)," August 2010.

Author's Address

	Ian Fette
	Google, Inc.
Email:	ifette+ietf@google.com
URI:	http://www.ianfette.com/