HyBi Working Group	I.F. Fette
Internet-Draft	Google, Inc.
Intended status: Standards Track	June 14, 2011
Expires: December 16, 2011	

The WebSocket protocol
draft-ietf-hybi-thewebsocketprotocol-09

Abstract

The WebSocket protocol enables two-way communication between a client 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 opening handshake followed by basic message framing, layered over TCP. (In theory, any transport protocol could be used so long as it provides for reliable transport, is byte clean, and supports relatively large message sizes. However, for this document, we consider only TCP.) The goal of this technology is to provide a mechanism for browser-based applications that need two-way communication with servers that does not rely on opening multiple HTTP connections (e.g. using XMLHttpRequest or <iframe>s and long polling).
Please send feedback to the hybi@ietf.org mailing list.

Status of this Memo

Copyright Notice

described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction

1.1. Background

This section is non-normative.
Historically, creating an instant messenger chat client as a Web application has required an abuse of HTTP to poll the server for updates while sending upstream notifications as distinct HTTP calls. [RFC6202]
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]
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 /chat HTTP/1.1
Host: server.example.com
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Key: dGhlIHNhbXBsZSBub25jZQ==
Sec-WebSocket-Origin: http://example.com
Sec-WebSocket-Protocol: chat, superchat
Sec-WebSocket-Version: 8
```

The handshake from the server looks as follows:

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: s3pPLMBiTxaQ9kYGzzhZRbK+xOo=
Sec-WebSocket-Protocol: chat
```

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 [RFC2616]. After the leading line in both cases come an unordered set of header fields. The meaning of these header fields is specified in Section 5 of this document. Additional header fields may also be present, such as cookies [I-D.ietf-httpstate-cookie] required to identify the user. The format and parsing of headers is as defined in [RFC2616].

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. The WebSocket message does not necessarily correspond to a particular network layer framing, as a fragmented message may be coalesced, or vice versa, e.g. by an intermediary.
Data is sent on the wire in the form of frames that have an associated type. A message is composed of one or more frames, all of which contain the same type of data. Broadly speaking, there are types for textual data, which is interpreted as UTF-8 [RFC3629] 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 signaling, such as to signal that the connection

should be closed. This version of the protocol defines six frame types and leaves ten reserved for future use.
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.

1.3. Opening Handshake

This section is non-normative.
The opening handshake is intended to be compatible with HTTP-based server-side software and intermediaries, 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 is an HTTP Upgrade request:

```
GET /chat HTTP/1.1
Host: server.example.com
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Key: dGhlIHNhbXBsZSBub25jZQ==
Sec-WebSocket-Origin: http://example.com
Sec-WebSocket-Protocol: chat, superchat
Sec-WebSocket-Version: 8
```

Headers in the handshake are sent by the client in a random order; the order is not meaningful.
The "Request-URI" of the GET method [RFC2616] is used to identify the endpoint of the WebSocket connection, both to allow multiple domains to be served from one IP address and to allow multiple WebSocket endpoints to be served by a single server.

The client includes the hostname in the Host header of its handshake as per [RFC2616], so that both the client and the server can verify that they agree on which host is in use.
Additional headers are used to select options in the WebSocket protocol. 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| request-header field can be used to indicate what subprotocols (application-level protocols layered over the WebSocket protocol) are acceptable to the client. The server selects one of the acceptable protocols and echoes that value in its handshake to indicate that it has selected that protocol.

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

The |Sec-WebSocket-Origin| header 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 is informed of the script origin generating the WebSocket connection request. If the server does not wish to accept connections from this origin, it can choose to reject the connection by sending an appropriate HTTP error code. This header is sent by browser clients, for non-browser clients this header may be sent if it makes sense in the context of those clients.

NOTE: It is worth noting that for the attack cases this header protects against, the untrusted party is typically the author of a JavaScript application that is executing in the context of the client. The client itself can contact the server and via the mechanism of the |Sec-WebSocket-Origin| header, determine whether to extend those communication privileges to the JavaScript application. A JavaScript application cannot set a header starting with "Sec-" via XHR. The intent is not to prevent non-browsers from establishing connections, but rather to ensure that browsers under the control of potentially malicious JavaScript cannot fake a WebSocket handshake.

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 two pieces of information and combine them to form a response. The first piece of information comes from the |Sec-WebSocket-Key| header in the client handshake:

```
Sec-WebSocket-Key: dGhlIHNhbXBsZSBub25jZQ==
```

For this header, the server has to take the value (as present in the header, e.g. the base64-encoded [RFC4648] version minus leading and trailing whitespace), and concatenate this with the GUID "258EAFA5-E914-47DA-95CA-C5AB0DC85B11" in string form, which is unlikely to be used by network endpoints that do not understand the WebSocket protocol. A SHA-1 hash (160 bits), base64-encoded, of this concatenation is then returned in the server's handshake [FIPS.180-2.2002].

Concretely, if as in the example above, header |Sec-WebSocket-Key| had the value "dGhlIHNhbXBsZSBub25jZQ==", the server would concatenate the string "258EAFA5-E914-47DA-95CA-C5AB0DC85B11" to form the string "dGhlIHNhbXBsZSBub25jZQ==258EAFA5-E914-47DA-95CA-C5AB0DC85B11". The server would then take the SHA-1 hash of this, giving the value 0xb3 0x7a 0x4f 0x2c 0xc0 0x62 0x4f 0x16 0x90 0xf6 0x46 0x06 0xcf 0x38 0x59 0x45 0xb2 0xbe 0xc4 0xea. This value is then base64-encoded, to give the value "s3pPLMBiTxaQ9kYGzzhZRbK+xOo=". This value would then be echoed in the header |Sec-WebSocket-Accept|.

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:

```
HTTP/1.1 101 Switching Protocols
```

Any status code other than 101 indicates that the WebSocket handshake has not completed, and that the semantics of HTTP still apply. The headers follow the status code.
The |Connection| and |Upgrade| headers complete the HTTP Upgrade. The |Sec-WebSocket-Accept| header indicates whether the server is willing to accept the connection. If present, this header must include a hash of the client's nonce sent in |Sec-WebSocket-Key| along with a predefined GUID. Any other value must not be interpreted as an acceptance of the connection by the server.

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: s3pPLMBiTxaQ9kYGzzhZRbK+xOo=
```

These fields are checked by the Web browser when it is acting as a |WebSocket| client for scripted pages. If the |Sec-WebSocket-Accept| value does not match the expected value, or if the header is missing, or if the HTTP status code is not 101, the connection will not be established and WebSocket frames will not be sent.
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 has selected. Web browsers verify that the server included one of the values as was specified in the WebSocket client's handshake. A server that speaks multiple subprotocols has to make sure it selects one based on the client's handshake and specifies it in its handshake.

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

The server can also set cookie-related option fields to *set* cookies, as in HTTP.

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 (detailed in Section 4.5.1). Upon receiving such a frame, the other peer sends a close frame

in response, 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 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 complement 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.
By sending a close frame and waiting for a close frame in response, certain cases are avoided where data may be unnecessarily lost. For instance, on some platforms, if a socket is closed with data in the receive queue, a RST packet is sent, which will then cause recv() to fail for the party that received the RST, even if there was data waiting to be read.

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 includes an additional closing handshake in-band that is designed to work in the presence of proxies and other intermediaries. 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 request mechanism also.
The protocol is intended to be extensible; future versions will likely introduce additional concepts such as multiplexing.

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 [RFC5321] and 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, at the time of writing of this specification, fields starting with |Sec-| cannot be set by an attacker from a Web browser using only HTML and JavaScript APIs such as |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.
By default the WebSocket protocol uses port 80 for regular WebSocket connections and port 443 for WebSocket connections tunneled over TLS [RFC2818].

1.8. Establishing a Connection

This section is non-normative.
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. At the time of writing of this specification, it should be noted that connections on port 80 and 443 have significantly different success rates, with connections on port 443 being significantly more likely to succeed, though this may change with time.

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 one of the selected subprotocol values in its response for the connection to be established.

These subprotocol names should be registered as per Section 11.10. To avoid potential collisions, it is recommended to use 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 Organization called their competing subprotocol "chat.example.org", 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, e.g. 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. [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 clients and servers.

2.1. Terminology

ASCII shall mean the character-encoding scheme defined in [ANSI.X3-4.1986].
This document makes reference to UTF-8 values and uses UTF-8 notational formats as defined in STD 63 [RFC3629].
Key Terms such as named algorithms or definitions are indicated like *this*.
Names of headers or variables are indicated like |this|.
Variable values are indicated like /this/.
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 "URI" is used in this document as defined in [RFC3986].
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 URIs

This specification defines two URI schemes, using the ABNF syntax defined in RFC 5234 [RFC5234], and terminology and ABNF productions defined by the URI specification RFC 3986 [RFC3986].

```
ws-URI = "ws:" "//" host [ ":" port ] path [ "?" query ]
wss-URI = "wss:" "//" host [ ":" port ] path [ "?" query ]

host = <host, defined in [RFC3986], Section 3.2.2>
port = <port, defined in [RFC3986], Section 3.2.3>
path = <path-abempty, defined in [RFC3986], Section 3.3>
query = <query, defined in [RFC3986], Section 3.4>
```

The port component is OPTIONAL; the default for "ws" is port 80, while the default for "wss" is port 443.
The URI is called "secure" if the scheme component matches "wss" case-insensitively.
The "resource-name" can be constructed by concatenating

*"/" if the path component is empty

*the path component

*"?" if the query component is non-empty

*the query component

Fragment identifiers are meaningless in the context of WebSocket URIs, and MUST NOT be used on these URIs. The character "#" in URIs MUST be escaped as %23 if used as part of the query component.

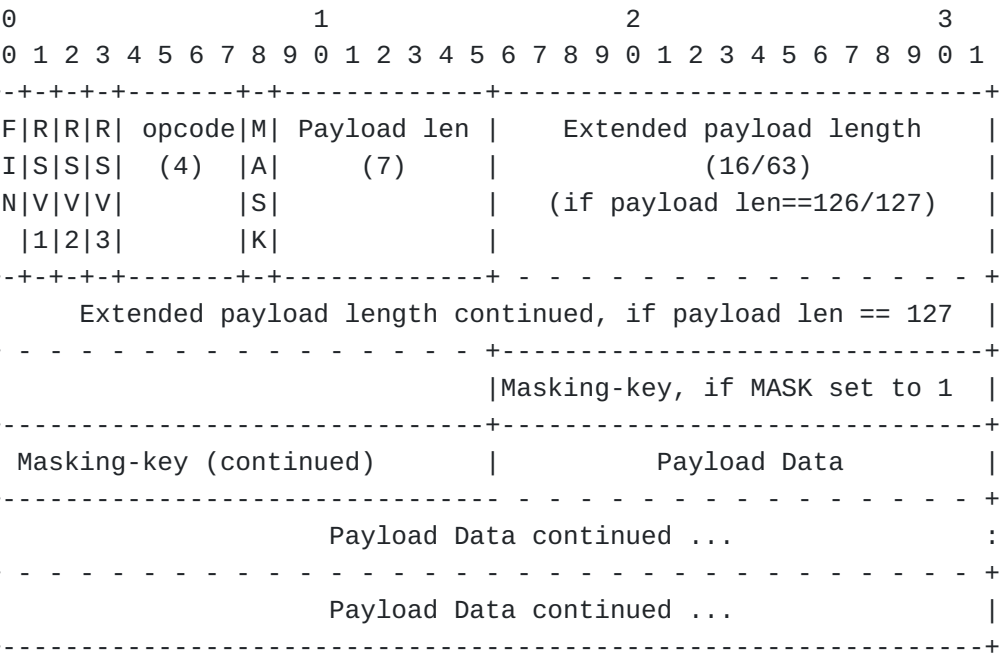
4. Data Framing

4.1. Overview

In the WebSocket protocol, data is transmitted using a sequence of frames. Frames sent from the client to the server are masked to avoid confusing network intermediaries, such as intercepting proxies. Frames sent from the server to the client are not masked.
The base framing protocol defines a frame type with an opcode, a payload length, and designated locations for extension and application data, which together define the *payload* data. Certain bits and opcodes are reserved for future expansion of the protocol.
A data frame MAY be transmitted by either the client or the server at any time after opening handshake completion and before that endpoint has sent a close frame (Section 4.5.1).

4.2. Base Framing Protocol

This wire format for the data transfer part is described by the ABNF [RFC5234] given in detail in this section. A high level overview of the framing is given in the following figure.



FIN:

1 bit

Indicates that this is the final fragment in a message. The first fragment MAY also be the final fragment.

RSV1, RSV2, RSV3: 1 bit each

MUST be 0 unless an extension is negotiated which defines meanings for non-zero values. If a nonzero value is received and none of the negotiated extensions defines the meaning of such a nonzero value, the receiving endpoint MUST *Fail the WebSocket Connection*.

Opcode: 4 bits

Defines the interpretation of the payload data. If an unknown opcode is received, the receiving endpoint MUST ignore that frame. The following values are defined.

*%x0 denotes a continuation frame

*%x1 denotes a text frame

*%x2 denotes a binary frame

*%x3-7 are reserved for further non-control frames

*%x8 denotes a connection close

*%x9 denotes a ping

*%xA denotes a pong

*%xB-F are reserved for further control frames

Mask: 1 bit

Defines whether the payload data is masked. If set to 1, a masking key is present in masking-key, and this is used to unmask the payload data as per Section 4.3. All frames sent from client to server have this bit set to 1.

Payload length: 7 bits, 7+16 bits, or 7+64 bits

The length of the payload data, in bytes: 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 most significant bit MUST be 0) are the payload length. Multibyte length quantities are expressed in network byte order. 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.

Masking-key: 0 or 4 bytes

All frames sent from the client to the server are masked by a 32-bit value that is contained within the frame. This field is present if the mask bit is set to 1, and is absent if the mask bit is set to 0. See Section 4.3 for further information on client-to-server masking.

Payload data: (x+y) bytes

The payload data is defined as extension data concatenated with application data.

Extension data: x bytes

The extension data is 0 bytes unless an extension has been negotiated. Any extension MUST specify the length of the extension data, or how that length may be calculated, and how the extension use MUST be negotiated during the opening handshake. If present, the extension data is included in the total payload length.

Application data: y 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.

The base framing protocol is formally defined by the following ABNF [RFC5234]:

```
ws-frame                = frame-fin
                          frame-rsv1
                          frame-rsv2
                          frame-rsv3
                          frame-opcode
                          frame-masked
                          frame-payload-length
                          [ frame-masking-key ]
                          frame-payload-data

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

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 ; text frame
                        / %x2 ; binary frame
                        / %x3-7 ; reserved for further non-control frames
                        / %x8 ; connection close
                        / %x9 ; ping
                        / %xA ; pong
                        / %xB-F ; reserved for further control frames

frame-masked            = %x0 ; frame is not masked, no frame-masking-key
                        / %x1 ; frame is masked, frame-masking-key present

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

frame-payload-length-16 = %x0000-FFFF

frame-payload-length-63 = %x0000000000000000-7FFFFFFFFFFFFFFF

frame-masking-key       = 4( %0x00-FF ) ; present only if frame-masked is 1

frame-payload-data      = (frame-masked-extension-data
                           frame-masked-application-data)   ; frame-masked 1
                        / (frame-unmasked-extension-data
                           frame-unmasked-application-data) ; frame-masked 0

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

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

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

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

4.3. Client-to-Server Masking

The client MUST mask all frames sent to the server. A server MUST close the connection upon receiving a frame with the MASK bit set to 0. In this case, a server MAY send a close frame with a status code of 1002 (protocol error) as defined in Section 7.4.1.
A masked frame MUST have the field frame-masked set to 1, as defined in Section 4.2.
The masking key is contained completely within the frame, as defined in Section 4.2 as frame-masking-key. It is used to mask the payload data defined in the same section as frame-payload-data, which includes extension and application data.
The masking key is a 32-bit value chosen at random by the client. The masking key MUST be derived from a strong source of entropy, and the masking key for a given frame MUST NOT make it simple for a server to predict the masking key for a subsequent frame. RFC 4086 [RFC4086] discusses what entails a suitable source of entropy for security-sensitive applications.
The masking does not affect the length of the payload data. To convert masked data into unmasked data, or vice versa, the following algorithm is applied. The same algorithm applies regardless of the direction of the translation - e.g. the same steps are applied to mask the data as to unmask the data.
Octet i of the transformed data ("transformed-octet-i") is the XOR of octet i of the original data ("original-octet-i") with octet i modulo 4 of the masking key ("masking-key-octet-j"):

```
j                   = i MOD 4
transformed-octet-i = original-octet-i XOR masking-key-octet-j
```

When preparing a masked frame, the client MUST pick a fresh masking key uniformly at random from the set of allowed 32-bit values. The unpredictability of the masking key is essential to prevent the author of malicious applications from selecting the bytes that appear on the wire.
The payload length, indicated in the framing as frame-payload-length, does NOT include the length of the masking key. It is the length of the payload data, e.g. the number of bytes following the masking key.

4.4. Fragmentation

The primary purpose of fragmentation is to allow sending a message that is of unknown size when the message is started without having to buffer that message. If messages couldn't be fragmented, then an endpoint

would have to buffer the entire message so its length could be counted before first byte is sent. With fragmentation, a server or intermediary may choose a reasonable size buffer, and when the buffer is full write a fragment to the network.
A secondary use-case for fragmentation is for multiplexing, where it is not desirable for a large message on one logical channel to monopolize the output channel, so the MUX needs to be free to split the message into smaller fragments to better share the output channel.
The following rules apply to fragmentation:

- An unfragmented message consists of a single frame with the FIN bit set and an opcode other than 0.

- A fragmented message consists of a single frame with the FIN bit clear and an opcode other than 0, followed by zero or more frames with the FIN bit clear and the opcode set to 0, and terminated by a single frame with the FIN bit set and an opcode of 0. A fragmented message is conceptually equivalent to a single larger message whose payload is equal to the concatenation of the payloads of the fragments in order, however in the presence of extensions this may not hold true as the extension defines the interpretation of the extension data present. For instance, extension data may only be present at the beginning of the first fragment and apply to subsequent fragments, or there may be extension data present in each of the fragments that applies only to that particular fragment. Setting aside the issue of extensions, the following example demonstrates how fragmentation works.

  EXAMPLE: For a text message sent as three fragments, the first fragment would have an opcode of 0x1 and a FIN bit clear, the second fragment would have an opcode of 0x0 and a FIN bit clear, and the third fragment would have an opcode of 0x0 and a FIN bit that is set.

- Control frames MAY be injected in the middle of a fragmented message. Control frames themselves MUST NOT be fragmented.

- Message fragments MUST be delivered to the recipient in the order sent by the sender.

- The fragments of one message MUST NOT be interleaved between the fragments of another message unless an extension has been negotiated that can interpret the interleaving.

- An endpoint MUST be capable of handling control frames in the middle of a fragmented message.

- A sender MAY create fragments of any size for non-control messages.

*Clients and servers MUST support receiving both fragmented and unfragmented messages.

*As control frames cannot be fragmented, an intermediary MUST NOT attempt to change the fragmentation of a control frame.

*An intermediary MUST NOT change the fragmentation of a message if any reserved bit values are used and the meaning of these values is not known to the intermediary.

*An intermediary MUST NOT change the fragmentation of any message in the context of a connection where extensions have been negotiated and the intermediary is not aware of the semantics of the negotiated extensions.

*As a consequence of these rules, all fragments of a message are of the same type, as set by the first fragment's opcode. Since Control frames cannot be fragmented, the type for all fragments in a message MUST be either text or binary, or one of the reserved opcodes.

Note: if control frames could not be interjected, the latency of a ping, for example, would be very long if behind a large message. Hence, the requirement of handling control frames in the middle of a fragmented message.

4.5. Control Frames

Control frames are identified by opcodes where the most significant bit of the opcode is 1. Currently defined opcodes for control frames include 0x8 (Close), 0x9 (Ping), and 0xA (Pong). Opcodes 0xB-0xF are reserved for further control frames yet to be defined.
Control frames are used to communicate state about the WebSocket.
Control frames can be interjected in the middle of a fragmented message.
All control frames MUST have a payload length of 125 bytes or less and MUST NOT be fragmented.

4.5.1. Close

The Close frame contains an opcode of 0x8.
The Close frame MAY contain a body (the "application data" portion of the frame) that indicates a reason for closing, such as an endpoint shutting down, an endpoint having received a frame too large, or an endpoint having received a frame that does not conform to the format expected by the other endpoint. If there is a body, the first two bytes of the body MUST be a 2-byte unsigned integer (in network byte order) representing a status code with value /code/ defined in Section 7.4.
Following the 2-byte integer the body MAY contain UTF-8 encoded data with value /reason/, the interpretation of which is not defined by this

specification. This data is not necessarily human readable, but may be useful for debugging or passing information relevant to the script that opened the connection.
Close frames sent from client to server must be masked as per Section 4.3.
The application MUST NOT send any more data frames after sending a close frame.
If an endpoint receives a Close frame and that endpoint did not previously send a Close frame, the endpoint MUST send a Close frame in response. It SHOULD do so as soon as is practical. An endpoint MAY delay sending a close frame until its current message is sent (for instance, if the majority of a fragmented message is already sent, an endpoint MAY send the remaining fragments before sending a Close frame). However, there is no guarantee that the endpoint which has already sent a Close frame will continue to process data.
After both sending and receiving a close message, an endpoint considers the WebSocket connection closed, and MUST close the underlying TCP connection. The server MUST close the underlying TCP connection immediately; the client SHOULD wait for the server to close the connection but MAY close the connection at any time after sending and receiving a close message, e.g. if it has not received a TCP close from the server in a reasonable time period.
If a client and server both send a Close message at the same time, both endpoints will have sent and received a Close message and should consider the WebSocket connection closed and close the underlying TCP connection.

4.5.2. Ping

The Ping frame contains an opcode of 0x9.
Upon receipt of a Ping frame, an endpoint MUST send a Pong frame in response. It SHOULD do so as soon as is practical. Pong frames are discussed in Section 4.5.3.
An endpoint MAY send a Ping frame any time after the connection is established and before the connection is closed. NOTE: A ping frame may serve either as a keepalive, or to verify that the remote endpoint is still responsive.

4.5.3. Pong

The Pong frame contains an opcode of 0xA.
Section 4.5.2 details requirements that apply to both Ping and Pong frames.
A Pong frame sent in response to a Ping frame must have identical Application Data as found in the message body of the Ping frame being replied to.
If an endpoint receives a Ping frame and has not yet sent Pong frame(s) in response to previous Ping frame(s), the endpoint MAY elect to send a Pong frame for only the most recently processed Ping frame.

A Pong frame MAY be sent unsolicited. This serves as a unidirectional heartbeat. A response to an unsolicited pong is not expected.

4.6. Data Frames

Data frames (e.g. non-control frames) are identified by opcodes where the most significant bit of the opcode is 0. Currently defined opcodes for data frames include 0x1 (Text), 0x2 (Binary). Opcodes 0x3-0x7 are reserved for further non-control frames yet to be defined.
Data frames carry application-layer or extension-layer data. The opcode determines the interpretation of the 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.

4.7. Examples

This section is non-normative.

*A single-frame unmasked text message

-0x81 0x05 0x48 0x65 0x6c 0x6c 0x6f (contains "Hello")

*A single-frame masked text message

-0x81 0x85 0x37 0xfa 0x21 0x3d 0x7f 0x9f 0x4d 0x51 0x58 (contains "Hello")

*A fragmented unmasked text message

-0x01 0x03 0x48 0x65 0x6c (contains "Hel")

-0x80 0x02 0x6c 0x6f (contains "lo")

*Ping request and response

-0x89 0x05 0x48 0x65 0x6c 0x6c 0x6f (contains a body of "Hello", but the contents of the body are arbitrary)

-0x8a 0x05 0x48 0x65 0x6c 0x6c 0x6f (contains a body of "Hello", matching the body of the ping)

*256 bytes binary message in a single unmasked frame

-0x82 0x7E 0x0100 [256 bytes of binary data]

*64KiB binary message in a single unmasked frame

-0x82 0x7F 0x0000000000010000 [65536 bytes of binary data]

4.8. Extensibility

The protocol is designed to allow for extensions, which will add capabilities to the base protocols. The endpoints of a connection MUST negotiate the use of any extensions during the opening handshake. This specification provides opcodes 0x3 through 0x7 and 0xB through 0xF, the extension data field, and the frame-rsv1, frame-rsv2, and frame-rsv3 bits of the frame header for use by extensions. The negotiation of extensions is discussed in further detail in Section 9.1. Below are some anticipated uses of extensions. This list is neither complete nor proscriptive.

*Extension data may be placed in the payload data 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 data to define larger opcodes or more per-frame bits.

5. Opening Handshake

5.1. Client Requirements

To *Establish a WebSocket Connection*, a client opens a connection and sends a handshake as defined in this section. A connection is defined to initially be in a CONNECTING state. A client will need to supply a /host/, /port/, /resource name/, and a /secure/ flag, which are the components of a WebSocket URI as discussed in Section 3, along with a list of /protocols/ and /extensions/ to be used. Additionally, if the client is a web browser, an /origin/ MUST be supplied.
Clients running in controlled environments, e.g. browsers on mobile handsets tied to specific carriers, may offload the management of the connection to another agent on the network. In such a situation, the client for the purposes of conformance is considered to include both the handset software and any such agents.

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

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

When the client is to *Establish a WebSocket Connection* given a set of (/host/, /port/, /resource name/, and /secure/ flag), along with a list of /protocols/ and /extensions/ to be used, and an /origin/ in the case of web browsers, it MUST open a connection, send an opening handshake, and read the server's handshake in response. The exact requirements of how the connection should be opened, what should be sent in the opening handshake, and how the server's response should be interpreted, are as follows in this section. In the following text, we will use terms from Section 3 such as "/host/" and "/secure/ flag" as defined in that section.

1. The components of the WebSocket URI passed into this algorithm (/host/, /port/, /resource name/ and /secure/ flag) MUST be valid according to the specification of WebSocket URIs specified in Section 3. If any of the components are invalid, the client MUST *Fail the WebSocket Connection* and abort these steps.

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

   If the client cannot determine the IP address of the remote host (for example because all communication is being done through a proxy server that performs DNS queries itself), then the client MUST assume for the purposes of this step that each host name refers to a distinct remote host, and should instead limit the total number of simultaneous connections that are not established to a reasonably low number (e.g., in a Web browser, simultaneous pending connections to a.example.com and b.example.com would be allowed, but if thirty connections are requested, that may not be allowed. The limit should consider the number of tabs the user has open.

   NOTE: This makes it harder for a script to perform a denial of service attack by just opening a large number of WebSocket connections to a remote host. A server can further reduce the load on itself when attacked by making use of this by pausing

before closing the connection, as that will reduce the rate at which the client reconnects.

NOTE: There is no limit to the number of established WebSocket connections a client can have with a single remote host. Servers can refuse to accept connections from hosts with an excessive number of existing connections, or disconnect resource-hogging connections when suffering high load.

3. *Proxy Usage*: If the client is configured to use a proxy when using the WebSocket protocol to connect to host /host/ and/or port /port/, then the client SHOULD 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 client 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:

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

   If the client is not configured to use a proxy, then a direct TCP connection SHOULD be opened 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 URI to pass the function MUST be constructed from /host/, /port/, / resource name/, and the /secure/ flag using the definition of a WebSocket URI as given in Section 3.

   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, either because a direct connection failed or because any proxy used returned an error, then the client MUST *Fail the WebSocket Connection* and abort the connection attempt.

5. If /secure/ is true, the client MUST perform a TLS handshake over the connection after opening the connection and before sending the handshake data [RFC2818]. If this fails (e.g. the server's certificate could not be verified), then the client MUST *Fail the WebSocket Connection* and abort the connection. Otherwise, all further communication on this channel MUST run through the encrypted tunnel. [RFC5246]

   Clients MUST use the Server Name Indication extension in the TLS handshake. [RFC6066]

Once a connection to the server has been established (including a connection via a proxy or over a TLS-encrypted tunnel), the client MUST send an opening handshake to the server. The handshake consists of an HTTP upgrade request, along with a list of required and optional headers. The requirements for this handshake are as follows.

1. The handshake MUST be a valid HTTP request as specified by [RFC2616].

2. The Method of the request MUST be GET and the HTTP version MUST be at least 1.1.

   For example, if the WebSocket URI is "ws://example.com/chat", The first line sent should be "GET /chat HTTP/1.1"

3. The request MUST contain a "Request-URI" as part of the GET method. This MUST match the /resource name/ Section 3 (a relative URI), or be an absolute URI that, when parsed, has a matching /resource name/ as well as matching /host/, /port/, and appropriate scheme (ws or wss).

4. The request MUST contain a "Host" header whose value is equal to /host/.

5. The request MUST contain an "Upgrade" header whose value is equal to "websocket".

6. The request MUST contain a "Connection" header whose value MUST include the "Upgrade" token.

7. The request MUST include a header with the name "Sec-WebSocket-Key". The value of this header MUST be a nonce consisting of a randomly selected 16-byte value that has been base64-encoded [RFC3548]. The nonce MUST be selected randomly for each connection.

   NOTE: As an example, if the randomly selected value was the sequence of bytes 0x01 0x02 0x03 0x04 0x05 0x06 0x07 0x08 0x09

0x0a 0x0b 0x0c 0x0d 0x0e 0x0f 0x10, the value of the header would be "AQIDBAUGBwgJCgsMDQ4PEC=="

8. The request MUST include a header with the name "Sec-WebSocket-Origin" if the request is coming from a browser client. If the connection is from a non-browser client, the request MAY include this header if the semantics of that client match the use-case described here for browser clients. The value of this header MUST be the ASCII serialization of origin of the context in which the code establishing the connection is running, and MUST be lower-case. The value MUST NOT contain letters in the range U+0041 to U+005A (i.e. LATIN CAPITAL LETTER A to LATIN CAPITAL LETTER Z) [I-D.ietf-websec-origin]. The ABNF is as defined in Section 6.1 of [I-D.ietf-websec-origin].

   As an example, if code is running on www.example.com attempting to establish a connection to ww2.example.com, the value of the header would be "http://www.example.com".

9. The request MUST include a header with the name "Sec-WebSocket-Version". The value of this header MUST be 8. *Note: Although a draft -09 was published, as -09 was comprised of editorial changes and not changes to the wire protocol, 9 was not used as a valid value for Sec-WebSocket-Version. This value was reserved in the IANA registry but was not and will not be used. If subsequent changes to the wire protocol are necessary, 9 will be skipped to prevent confusion with the draft 9 protocol.*

10. The request MAY include a header with the name "Sec-WebSocket-Protocol". If present, this value indicates the subprotocol(s) the client wishes to speak, ordered by preference. The elements that comprise this value MUST be non-empty strings with characters in the range U+0021 to U+007E not including separator characters as defined in [RFC2616], and MUST all be unique strings. The ABNF for the value of this header is 1#token, where the definitions of constructs and rules are as given in [RFC2616].

11. The request MAY include a header with the name "Sec-WebSocket-Extensions". If present, this value indicates the protocol-level extension(s) the client wishes to speak. The interpretation and format of this header is described in Section 9.1.

12. The request MAY include headers associated with sending cookies, as defined by the appropriate specifications [I-D.ietf-httpstate-cookie]. These headers are referred to as *Headers to Send Appropriate Cookies*.

Once the client's opening handshake has been sent, the client MUST wait for a response from the server before sending any further data. The client MUST validate the server's response as follows:

1. If the status code received from the server is not 101, the client handles the response per HTTP procedures. Otherwise, proceed as follows.

2. If the response lacks an "Upgrade" header or the "Upgrade" header contains a value that is not an ASCII case-insensitive match for the value "websocket", the client MUST *Fail the WebSocket Connection* .

3. If the response lacks a "Connection" header or the "Connection" header contains a value that is not an ASCII case-insensitive match for the value "Upgrade", the client MUST *Fail the WebSocket Connection*.

4. If the response lacks a "Sec-WebSocket-Accept" header or the "Sec-WebSocket-Accept" contains a value other than the base64-encoded SHA-1 of the concatenation of the "Sec-WebSocket-Key" (as a string, not base64-decoded) with the string "258EAFA5-E914-47DA-95CA-C5AB0DC85B11", the client MUST *Fail the WebSocket Connection*

5. If the response includes a "Sec-WebSocket-Extensions" header, and this header indicates the use of an extension that was not present in the client' handshake (the server has indicated an extension not requested by the client), the client MUST *Fail the WebSocket Connection*. (The parsing of this header to determine which extensions are requested is discussed in Section 9.1.)

If the server's response is validated as provided for above, it is said that *The WebSocket Connection is Established* and that the WebSocket Connection is in the OPEN state. The *Extensions In Use* is defined to be a (possibly empty) string, the value of which is equal to the value of the |Sec-WebSocket-Extensions| header supplied by the server's handshake, or the null value if that header was not present in the server's handshake. The *Subprotocol In Use* is defined to be the value of the |Sec-WebSocket-Protocol| header in the server's handshake, or the null value if that header was not present in the server's handshake. Additionally, if any headers in the server's handshake indicate that cookies should be set (as defined by [I-D.ietf-httpstate-cookie]), these cookies are referred to as *Cookies Set During the Server's Opening Handshake*.

5.2. Server-side Requirements

This section only applies to servers.

Servers MAY offload the management of the connection to other agents on the network, for example load balancers and reverse proxies. In such a situation, the server for the purposes of conformance is considered to include all parts of the server-side infrastructure from the first device to terminate the TCP connection all the way to the server that processes requests and sends responses.
EXAMPLE: For example, a data center might have a server that responds to WebSocket requests with an appropriate handshake, and then passes the connection to another server to actually process the data frames. For the purposes of this specification, the "server" is the combination of both computers.

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's opening 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 MUST stop processing the client's handshake, and return an HTTP response with an appropriate error code (such as 400 Bad Request).

1. An HTTP/1.1 or higher GET request, including a "Request-URI" [RFC2616] that should be interpreted as a /resource name/ Section 3.

2. A "Host" header containing the server's authority.

3. A "Sec-WebSocket-Key" header with a base64-encoded value that, when decoded, is 16 bytes in length.

4. A "Sec-WebSocket-Version" header, with a value of 8.

5. Optionally, a "Sec-WebSocket-Origin" header. This header is sent by all browser clients. A connection attempt lacking this header SHOULD NOT be interpreted as coming from a browser client.

6. Optionally, a "Sec-WebSocket-Protocol" header, with a list of values indicating which protocols the client would like to speak, ordered by preference.

7. Optionally, a "Sec-WebSocket-Extensions" header, with a list of values indicating which extensions the client would like to speak. The interpretation of this header is discussed in Section 9.1.

8. Optionally, other headers, such as those used to send cookies to a server. Unknown headers MUST be ignored.

5.2.2. Sending the Server's Opening Handshake

```
accept-value     = base64-value
base64-value     = *base64-data [ base64-padding ]
base64-data      = 4base64-character
base64-padding   = (2base64-character "==") / (3base64-character "=")
base64-character = ALPHA / DIGIT / "+" / "/"
```

When a client establishes a WebSocket connection to a server, the server MUST complete the following steps to accept the connection and send the server's opening handshake.

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's handshake) MUST run through the encrypted tunnel. [RFC5246]

2. Establish the following information:

   /origin/
   The |Sec-WebSocket-Origin| header in the client's handshake indicates the origin of the script establishing the connection. The origin is serialized to ASCII and converted to lowercase. The server MAY use this information as part of a determination of whether to accept the incoming connection. If the server does not validate the origin, it will accept connections from anywhere. If the server does not wish to accept this connection, it MUST return an appropriate HTTP error code (e.g. 403 Forbidden) and abort the WebSocket handshake described in this section. For more detail, refer to Section 10.

   /key/
   The |Sec-WebSocket-Key| header in the client's handshake includes a base64-encoded value that, if decoded, is 16 bytes in length. This (encoded) value is used in the creation of the server's handshake to indicate an acceptance of the connection. It is not necessary for the server to base64-decode the "Sec-WebSocket-Key" value.

   /version/
   The |Sec-WebSocket-Version| header in the client's handshake includes the version of the WebSocket protocol the

client is attempting to communicate with. If this version does not match a version understood by the server, the server MUST abort the websocket handshake described in this section and instead send an appropriate HTTP error code (such as 426 Upgrade Required), and a |Sec-WebSocket-Version| header indicating the version(s) the server is capable of understanding.

/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 from the Request-URI [RFC2616] of the GET method. If the requested service is not available, the server MUST send an appropriate HTTP error code (such as 404 Not Found) and abort the WebSocket handshake.

/subprotocol/
Either a single value or null, representing the subprotocol the server is ready to use. If the server supports multiple subprotocols, then the value MUST be derived from the client's handshake, specifically by selecting one of the values 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, and is not a legal value for this field. The ABNF for the value of this header is (token), where the definitions of constructs and rules are as given in [RFC2616].

/extensions/
A (possibly empty) list representing the protocol-level extensions the server is ready to use. If the server supports multiple extensions, then the value MUST be derived from the client's handshake, specifically by selecting one or more of the values from the "Sec-WebSocket-Extensions" 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. Extensions not listed by the client MUST NOT be listed. The method by which these values should be selected and interpreted is discussed in Section 9.1.

3. If the server chooses to accept the incoming connection, it MUST reply with a valid HTTP response indicating the following.

    1. A Status-Line with a 101 response code as per RFC 2616 [RFC2616]. Such a response could look like "HTTP/1.1 101 Switching Protocols"

2. An "Upgrade" header with value "websocket" as per RFC 2616 [RFC2616].

3. A "Connection" header with value "Upgrade"

4. A "Sec-WebSocket-Accept" header. The value of this header is constructed by concatenating /key/, defined above in [server_handshake_info] of Section 5.2.2, with the string "258EAFA5-E914-47DA-95CA-C5AB0DC85B11", taking the SHA-1 hash of this concatenated value to obtain a 20-byte value, and base64-encoding this 20-byte hash.

   The ABNF of this header is defined as follows:

   NOTE: As an example, if the value of the "Sec-WebSocket-Key" header in the client's handshake were "dGhlIHNhbXBsZSBub25jZQ==", the server would append the string "258EAFA5-E914-47DA-95CA-C5AB0DC85B11" to form the string "dGhlIHNhbXBsZSBub25jZQ==258EAFA5-E914-47DA-95CA-C5AB0DC85B11". The server would then take the SHA-1 hash of this string, giving the value 0xb3 0x7a 0x4f 0x2c 0xc0 0x62 0x4f 0x16 0x90 0xf6 0x46 0x06 0xcf 0x38 0x59 0x45 0xb2 0xbe 0xc4 0xea. This value is then base64-encoded, to give the value "s3pPLMBiTxaQ9kYGzzhZRbK+xOo=", which would be returned in the "Sec-WebSocket-Accept" header.

5. Optionally, a "Sec-WebSocket-Protocol" header, with a value /subprotocol/ as defined in [server_handshake_info] of Section 5.2.2.

6. Optionally, a "Sec-WebSocket-Extensions" header, with a value /extensions/ as defined in [server_handshake_info] of Section 5.2.2. If multiple extensions are to be used, they must all be listed in a single Sec-WebSocket-Extensions header. This header MUST NOT be repeated.

This completes the server's handshake. If the server finishes these steps without aborting the WebSocket handshake, the server considers the WebSocket connection to be established and that the WebSocket connection is in the OPEN state. At this point, the server may begin sending (and receiving) data.

6. Sending and Receiving Data

6.1. Sending Data

To *Send a WebSocket Message* comprising of /data/ over a WebSocket connection, an endpoint MUST perform the following steps.

1. The endpoint MUST ensure the WebSocket connection is in the OPEN state (cf. Section 5.1 and Section 5.2.2.) If at any point the state of the WebSocket connection changes, the endpoint MUST abort the following steps.

2. An endpoint MUST encapsulate the /data/ in a WebSocket frame as defined in Section 4.2. If the data to be sent is large, or if the data is not available in its entirety at the point the endpoint wishes to begin sending the data, the endpoint MAY alternately encapsulate the data in a series of frames as defined in Section 4.4.

3. The opcode (frame-opcode) of the first frame containing the data MUST be set to the appropriate value from Section 4.2 for data that is to be interpreted by the recipient as text or binary data.

4. The FIN bit (frame-fin) of the last frame containing the data MUST be set to 1 as defined in Section 4.2.

5. If the data is being sent by the client, the frame(s) MUST be masked as defined in Section 4.3.

6. If any extensions (Section 9) have been negotiated for the WebSocket connection, additional considerations may apply as per the definition of those extensions.

7. The frame(s) that have been formed MUST be transmitted over the underlying network connection.

6.2. Receiving Data

To receive WebSocket data, an endpoint listens on the underlying network connection. Incoming data MUST be parsed as WebSocket frames as defined in Section 4.2. If a control frame (Section 4.5) is received, the frame MUST be handled as defined by Section 4.5. Upon receiving a data frame (Section 4.6), the endpoint MUST note the /type/ of the data as defined by the Opcode (frame-opcode) from Section 4.2. The *Application Data* from this frame is defined as the /data/ of the message. If the frame comprises an unfragmented message (Section 4.4), it is said that *A WebSocket Message Has Been Received* with type /type/ and data /data/. If the frame is part of a fragmented message, the *Application Data* of the subsequent data frames is concatenated to form

the /data/. When the last fragment is received as indicated by the FIN bit (frame-fin), it is said that *A WebSocket Message Has Been Received* with data /data/ (comprised of the concatenation of the *Application Data* of the fragments) and type /type/ (noted from the first frame of the fragmented message). Subsequent data frames MUST be interpreted as belonging to a new WebSocket Message.

Extensions (Section 9) MAY change the semantics of how data is read, specifically including what comprises a message boundary. Extensions, in addition to adding "Extension data" before the "Application data" in a payload, MAY also modify the "Application data" (such as by compressing it).

Data frames received by a server from a client MUST be unmasked as described in Section 4.3.

7. Closing the connection

7.1. Definitions

7.1.1. Close the WebSocket Connection

To *Close the WebSocket Connection*, an endpoint closes the underlying TCP connection. An endpoint SHOULD use a method that cleanly closes the TCP connection, as well as the TLS session, if applicable, discarding any trailing bytes that may be received. An endpoint MAY close the connection via any means available when necessary, such as when under attack.

The underlying TCP connection, in most normal cases, SHOULD be closed first by the server, so that it holds the TIME_WAIT state and not the client (as this would prevent it from re-opening the connection for 2 MSL, while there is no corresponding server impact as a TIME_WAIT connection is immediately reopened upon a new SYN with a higher seq number). In abnormal cases (such as not having received a TCP Close from the server after a reasonable amount of time) a client MAY initiate the TCP Close. As such, when a server is instructed to *Close the WebSocket Connection* it SHOULD initiate a TCP Close immediately, and when a client is instructed to do the same, it SHOULD wait for a TCP Close from the server.

As an example of how to obtain a clean closure in C using Berkeley sockets, one would call shutdown() with SHUT_WR on the socket, call recv() until obtaining a return value of 0 indicating that the peer has also performed an orderly shutdown, and finally calling close() on the socket.

7.1.2. Start the WebSocket Closing Handshake

To *Start the WebSocket Closing Handshake* with a status code (Section 7.4) /code/ and an optional close reason (Section 7.1.6) /reason/, an endpoint MUST send a Close control frame, as described in Section 4.5.1 whose status code is set to /code/ and whose close reason is set to /

reason/. Once an endpoint has both sent and received a Close control frame, that endpoint SHOULD *Close the WebSocket Connection* as defined in Section 7.1.1.

7.1.3. The WebSocket Closing Handshake is Started

Upon either sending or receiving a Close control frame, it is said that *The WebSocket Closing Handshake is Started* and that the WebSocket connection is in the CLOSING state.

7.1.4. The WebSocket Connection is Closed

When the underlying TCP connection is closed, it is said that *The WebSocket Connection is Closed* and that the WebSocket connection is in the CLOSED state. If the tcp connection was closed after the WebSocket closing handshake was completed, the WebSocket connection is said to have been closed *cleanly*.
If the WebSocket connection could not be established, it is also said that *The WebSocket Connection is Closed*, but not cleanly.

7.1.5. The WebSocket Connection Close Code

As defined in Section 4.5.1 and Section 7.4, a Close control frame may contain a status code indicating a reason for closure. A closing of the WebSocket connection may be initiated by either endpoint, potentially simultaneously. *The WebSocket Connection Close Code* is defined as the status code (Section 7.4) contained in the first Close control frame received by the application implementing this protocol. If this Close control frame contains no status code, *The WebSocket Connection Close Code* is considered to be 1005. If *The WebSocket Connection is Closed* and no Close control frame was received by the endpoint (such as could occur if the underlying transport connection is lost), *The WebSocket Connection Close Code* is considered to be 1006.
NOTE: Two endpoints may not agree on the value of *The WebSocket Connection Close Code*. As an example, if the remote endpoint sent a Close frame but the local application has not yet read the data containing the Close frame from its socket's receive buffer, and the local application independently decided to close the connection and send a Close frame, both endpoints will have sent and received a Close frame, and will not send further Close frames. Each endpoint will see the Connection Close Code sent by the other end as the *WebSocket Connection Close Code*. As such, it is possible that the two endpoints may not agree on the value of *The WebSocket Connection Close Code* in the case that both endpoints *Start the WebSocket Closing Handshake* independently and at roughly the same time.

7.1.6. The WebSocket Connection Close Reason

As defined in Section 4.5.1 and Section 7.4, a Close control frame may contain a status code indicating a reason for closure, followed by

UTF-8 encoded data, the interpretation of said data being left to the endpoints and not defined by this protocol. A closing of the WebSocket connection may be initiated by either endpoint, potentially simultaneously. *The WebSocket Connection Close Reason* is defined as the UTF-8 encoded data following the status code (Section 7.4) contained in the first Close control frame received by the application implementing this protocol. If there is no such data in the Close control frame, *The WebSocket Connection Close Reason* is the empty string.
NOTE: Following the same logic as noted in Section 7.1.5, two endpoints may not agree on *The WebSocket Connection Close Reason*.

7.1.7. Fail the WebSocket Connection

Certain algorithms and specifications require an endpoint to *Fail the WebSocket Connection*. To do so, the client MUST *Close the WebSocket Connection*, and MAY report the problem to the user (which would be especially useful for developers) in an appropriate manner.
If *The WebSocket Connection is Established* prior to the point where the endpoint is required to *Fail the WebSocket Connection*, the endpoint SHOULD send a Close frame with an appropriate status code Section 7.4 before proceeding to *Close the WebSocket Connection*. An endpoint MAY omit sending a Close frame if it believes the other side is unlikely to be able to receive and process the close frame, due to the nature of the error that led to the WebSocket connection being failed in the first place. An endpoint MUST NOT continue to attempt to process data (including a responding Close frame) from the remote endpoint after being instruted to *Fail the WebSocket Connection*.
Except as indicated above or as specified by the application layer (e.g. a script using the WebSocket API), clients SHOULD NOT close the connection.

7.2. Abnormal Closures

7.2.1. Client-Initiated Closure

Certain algorithms, namely during the opening handshake, require the client to *Fail the WebSocket Connection*. To do so, the client MUST *Fail the WebSocket Connection* as defined in Section 7.1.7.
If at any point the underlying transport layer connection is unexpectedly lost, the client MUST *Fail the WebSocket Connection*.
Except as indicated above or as specified by the application layer (e.g. a script using the WebSocket API), clients SHOULD NOT close the connection.

7.2.2. 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* (Section 7.1.1).

7.3. Normal Closure of Connections

Servers MAY close the WebSocket connection whenever desired. Clients SHOULD NOT close the WebSocket connection arbitrarily. In either case, an endpoint initiates a closure by following the procedures to *Start the WebSocket Closing Handshake* (Section 7.1.2).

7.4. Status Codes

When closing an established connection (e.g. when sending a Close frame, after the opening handshake has completed), an endpoint MAY indicate a reason for closure. The interpretation of this reason by an endpoint, and the action an endpoint should take given this reason, are left undefined by this specification. This specification defines a set of pre-defined status codes, and specifies which ranges may be used by extensions, frameworks, and end applications. The status code and any associated textual message are optional components of a Close frame.

7.4.1. Defined Status Codes

Endpoints MAY use the following pre-defined status codes when sending a Close frame.

1000

> 1000 indicates a normal closure, meaning whatever purpose the connection was established for has been fulfilled.

1001

> 1001 indicates that an endpoint is "going away", such as a server going down, or a browser having navigated away from a page.

1002

> 1002 indicates that an endpoint is terminating the connection due to a protocol error.

1003

> 1003 indicates that an endpoint is terminating the connection because it has received a type of data it cannot accept (e.g. an endpoint that understands only text data MAY send this if it receives a binary message).

1004

1004 indicates that an endpoint is terminating the connection because it has received a frame that is too large.

1005

1005 is a reserved value and MUST NOT be set as a status code in a Close control frame by an endpoint. It is designated for use in applications expecting a status code to indicate that no status code was actually present.

1006

1006 is a reserved value and MUST NOT be set as a status code in a Close control frame by an endpoint. It is designated for use in applications expecting a status code to indicate that the connection was closed abnormally, e.g. without sending or receiving a Close control frame.

7.4.2. Reserved Status Code Ranges

0-999

Status codes in the range 0-999 are not used.

1000-1999

Status codes in the range 1000-1999 are reserved for definition by this protocol.

2000-2999

Status codes in the range 2000-2999 are reserved for use by extensions.

3000-3999

Status codes in the range 3000-3999 MAY be used by libraries and frameworks. The interpretation of these codes is undefined by this protocol. End applications MUST NOT use status codes in this range.

4000-4999

Status codes in the range 4000-4999 MAY be used by application code. The interpretation of these codes is undefined by this protocol.

8. Error Handling

8.1. 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.

8.2. 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.

9. Extensions

WebSocket clients MAY request extensions to this specification, and WebSocket servers MAY accept some or all extensions requested by the client. A server MUST NOT respond with any extension not requested by the client. If extension parameters are included in negotiations between the client and the server, those parameters MUST be chosen in accordance with the specification of the extension to which the parameters apply.

9.1. Negotiating Extensions

A client requests extensions by including a "Sec-WebSocket-Extensions" header, which follows the normal rules for HTTP headers (see [RFC2616] section 4.2) and the value of the header is defined by the following ABNF. Note that unlike other section of the document this section is using ABNF syntax/rules from [RFC2616]. If a value is received by either the client or the server during negotiation that does not conform to the ABNF below, the recipient of such malformed data MUST immediately *Fail the WebSocket Connection*.

```
extension-list = 1#extension
extension = extension-token *( ";" extension-param )
extension-token = registered-token / private-use-token
registered-token = token
private-use-token = "x-" token
extension-param = token [ "=" token ]
```

Note that like other HTTP headers, this header MAY be split or combined across multiple lines. Ergo, the following are equivalent:

```
Sec-WebSocket-Extensions: foo
Sec-WebSocket-Extensions: bar; baz=2
```

is exactly equivalent to

```
Sec-WebSocket-Extensions: foo, bar; baz=2
```

Any extension-token used MUST either be a registered token (registration TBD), or have a prefix of "x-" to indicate a private-use token. The parameters supplied with any given extension MUST be defined for that extension. Note that the client is only offering to use any advertised extensions, and MUST NOT use them unless the server indicates that it wishes to use the extension.
Note that the order of extensions is significant. Any interactions between multiple extensions MAY be defined in the documents defining the extensions. In the absence of such definition, the interpretation is that the headers listed by the client in its request represent a preference of the headers it wishes to use, with the first options listed being most preferable. The extensions listed by the server in response represent the extensions actually in use for the connection. Should the extensions modify the data and/or framing, the order of operations on the data should be assumed to be the same as the order in which the extensions are listed in the server's response in the opening handshake.
For example, if there are two extensions "foo" and "bar", if the header |Sec-WebSocket-Extensions| sent by the server has the value "foo, bar" then operations on the data will be made as bar(foo(data)), be those changes to the data itself (such as compression) or changes to the framing thay may "stack".
Non-normative examples of acceptable extension headers:

```
Sec-WebSocket-Extensions: deflate-stream
Sec-WebSocket-Extensions: mux; max-channels=4; flow-control, deflate-stream
Sec-WebSocket-Extensions: x-private-extension
```

A server accepts one or more extensions by including a |Sec-WebSocket-Extensions| header containing one or more extensions which were requested by the client. The interpretation of any extension parameters, and what constitutes a valid response by a server to a requested set of parameters by a client, will be defined by each such extension.

9.2. Known Extensions

Extensions provide a mechanism for implementations to opt-in to additional protocol features. This section defines the meaning of well-known extensions but implementations MAY use extensions defined separately as well.

9.2.1. Compression

The registered extension token for this compression extension is "deflate-stream".

The extension does not have any per frame extension data and it does not define the use of any WebSocket reserved bits or opcodes.
Senders using this extension MUST apply [RFC1951] encodings to all bytes of the data stream following the opening handshake including both data and control frames. The data stream MAY include multiple blocks of both compressed and uncompressed types as defined by [RFC1951].
Senders MUST NOT delay the transmission of any portion of a WebSocket frame because the deflate encoding of the frame does not end on a byte boundary. The encodings for adjacent frames MAY appear in the same byte if no delay in transmission is occurred by doing so.
Historically there have been some confusion and interoperability problems around the specification of compression algorithms. In this specification "deflate-stream" requires a [RFC1951] deflate encoding. It MUST NOT be wrapped in any of the header formats often associated with RFC 1951 such as "zlib" [RFC1950]. This requirement is given special attention with this note because of confusion in this area, the presence of some popular open source libraries that create both formats under a single API call with confusing naming conventions, and the fact that the popular HTTP [RFC2616] specification defines "deflate" compression differently than this specification.

10. 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 an opening 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.

A common class of security problems arise when sending text data using using the wrong encoding. This protocol specifies that messages with a Text data type (as opposed to Binary or other types) contain UTF-8 encoded data. Although the length is still indicated and applications implementing this protocol should use the length to determine where the frame actually ends, sending data in an improper encoding may still break assumptions applications built on top of this protocol may make, leading from anything to misinterpretation of data to loss of data to potential security bugs.

In addition to endpoints being the target of attacks via WebSockets, other parts of web infrastructure, such as proxies, may be the subject of an attack. In particular, an intermediary may interpret a WebSocket frame from a client as a request, and a frame from the server as a response to that request. For instance, an attacker could get a browser to establish a connection to its server, get the browser to send a frame that looks to an intermediary like a GET request for a common piece of JavaScript on another domain, and send back a frame that is interpreted as a cacheable response to that request, thus poisioning the cache for other users. To prevent this attack, frames sent from clients are masked on the wire with a 32-bit value, to prevent an attacker from controlling the bits on the wire and thus lessen the probability of an attacker being able to construct a frame that can be misinterpreted by a proxy as a non-WebSocket request.

As mentioned in Section 8.2, servers must be extremely cautious interpreting invalid UTF-8 data from the client. A naive UTF-8 parsing implementation can result in buffer overflows in the case of invalid input data.

For connections using TLS (wss: URIs), the amount of benefit provided by TLS depends greatly on the strength of the algorithms negotiated during the TLS handshake. To achieve reasonable levels of protections, clients should use only Strong TLS algorithms. "Web Security Context: User Interface Guidelines" [W3C.REC-wsc-ui-20100812] discusses what constitutes Strong TLS algorithms.

11. IANA Considerations

11.1. Registration of "ws:" Scheme

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

A |ws:| URI 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][RFC3986]

The <path> [RFC3986] 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]

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 specifications. [RFC3986] [RFC3987]

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

Interoperability considerations.
None.

Security considerations.
See "Security considerations" section above.

Contact.
HYBI WG <hybi@ietf.org>

Author/Change controller.
IETF <iesg@ietf.org>

References.
RFC XXXX

11.2. Registration of "wss:" Scheme

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

A |wss:| URI identifies a WebSocket server and resource name, and indicates that traffic over that connection is to be protected via TLS (including standard benefits of TLS such as confidentiality, integrity, and authentication).

URI scheme name.
wss

Status.
Permanent.

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

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]

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

Applications/protocols that use this URI scheme name.

WebSocket protocol over TLS.

Interoperability considerations.

None.

Security considerations.

See "Security considerations" section above.

Contact.

HYBI WG <hybi@ietf.org>

Author/Change controller.

IETF <iesg@ietf.org>

References.

RFC XXXX

11.3. Registration of the "WebSocket" HTTP Upgrade Keyword

This section defines a keyword for registration in the "HTTP Upgrade Tokens" registry as per RFC 2817 [RFC2817].

Name of token.

WebSocket

Author/Change controller.

IETF <iesg@ietf.org>

Contact.

HYBI <hybi@ietf.org>

References.

RFC XXXX

11.4. Sec-WebSocket-Key

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

Header field name
Sec-WebSocket-Key

Applicable protocol
http

Status
standard

Author/Change controller
IETF

Specification document(s)
RFC XXXX

Related information
This header field is only used for WebSocket opening handshake.

The |Sec-WebSocket-Key| header is used in the WebSocket opening handshake. It is 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 opening handshake. This helps ensure that the server does not accept connections from non-WebSocket clients (e.g. HTTP clients) that are being abused to send data to unsuspecting WebSocket servers.

11.5. Sec-WebSocket-Extensions

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

Header field name
Sec-WebSocket-Extensions

Applicable protocol
http

Status
standard

Author/Change controller
IETF

Specification document(s)
RFC XXXX

Related information

This header field is only used for WebSocket opening handshake.

The |Sec-WebSocket-Extensions| header is used in the WebSocket opening handshake. It is initially sent from the client to the server, and then subsequently sent from the server to the client, to agree on a set of protocol-level extensions to use for the duration of the connection.

11.6. WebSocket Extension Name Registry

This specification requests the creation of a new IANA registry for WebSocket Extension names to be used with the WebSocket protocol in accordance with the principles set out in RFC 5226 [RFC5226].
As part of this registry IANA will maintain the following information:

Extension Identifier
The identifier of the extension, as will be used in the Sec-WebSocket-Extension header registered in Section 11.5 of this specification. The value must conform to the requirements for an extension-token as defined in Section 9.1 of this specification.

Extension Common Name
The name of the extension, as the extension is generally referred to.

Extension Definition
A reference to the document in which the extension being used with the WebSocket protocol is defined.

Known Incompatible Extensions
A list of extension identifiers with which this extension is known to be incompatible.

WebSocket Extension names are to be subject to First Come First Serve as per RFC5226 [RFC5226], with the exception of WebSocket Extension names whose Extension Identifier matches a private-use-token as defined in Section 9.1 (values beginning with "x-"). These Extension Identifiers matching private-use-token are reserved for Experimental Use as per RFC5226 [RFC5226].

Extension Identifier	Extension Common Name	Extension Definition
deflate-stream	Deflate Stream Compression	Section 9.2.1 of this document.

IANA is asked to add an initial value to the registry (there are no known incompatible extensions for this initial entry):

11.7. Sec-WebSocket-Accept

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

Header field name
Sec-WebSocket-Accept

Applicable protocol
http

Status
standard

Author/Change controller
IETF

Specification document(s)
RFC XXXX

Related information
This header field is only used for WebSocket opening handshake.

The |Sec-WebSocket-Accept| header is used in the WebSocket opening handshake. It is sent from the server to the client to confirm that the server is willing to initiate the connection.

11.8. Sec-WebSocket-Origin

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

Header field name
Sec-WebSocket-Origin

Applicable protocol
http

Status
standard

Author/Change controller
IETF

Specification document(s)
RFC XXXX

Related information
This header field is only used for WebSocket opening handshake.

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

11.9. Sec-WebSocket-Protocol

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

Header field name
Sec-WebSocket-Protocol

Applicable protocol
http

Status
standard

Author/Change controller
IETF

Specification document(s)
RFC XXXX

Related information
This header field is only used for WebSocket opening handshake.

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

11.10. WebSocket Subprotocol Name Registry

This specification requests the creation of a new IANA registry for WebSocket Subprotocol names to be used with the WebSocket protocol in accordance with the principles set out in RFC 5226 [RFC5226].
As part of this registry IANA will maintain the following information:

Subprotocol Identifier
The identifier of the subprotocol, as will be used in the Sec-WebSocket-Protocol header registered in Section 11.9 of this specification. The value must conform to the requirements given in [swp-defined] of Section 5.1, Paragraph 4 of Section 5.1 of this specification, namely the value must be a token as defined by RFC 2616 [RFC2616].

Subprotocol Common Name

The name of the subprotocol, as the subprotocol is generally referred to.

Subprotocol Definition

A reference to the document in which the subprotocol being used with the WebSocket protocol is defined.

WebSocket Subprotocol names are to be subject to First Come First Serve as per RFC5226 [RFC5226].

11.11. Sec-WebSocket-Version

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

Header field name

Sec-WebSocket-Version

Applicable protocol

http

Status

standard

Author/Change controller

IETF

Specification document(s)

RFC XXXX

Related information

This header field is only used for WebSocket opening handshake.

The |Sec-WebSocket-Version| header is used in the WebSocket opening handshake. It is sent from the client to the server to indicate the protocol version of the connection. This enables servers to correctly interpret the opening handshake and subsequent data being sent from the data, and close the connection if the server cannot interpret that data in a safe manner.

11.12. WebSocket Version Number Registry

This specification requests the creation of a new IANA registry for WebSocket Version Numbers to be used with the WebSocket protocol in accordance with the principles set out in RFC 5226 [RFC5226].
As part of this registry IANA will maintain the following information:

Version Number

The version number to be used in the Sec-WebSocket-Version as specified in Section 5.1 of this specification. The value must be a whole number (zero or higher).

Reference

The RFC requesting a new version number.

WebSocket Version Numbers are to be subject to RFC Required as per RFC5226 [RFC5226].

Version Number	Reference
0	draft-ietf-hybi-thewebsocketprotocol-00
1	draft-ietf-hybi-thewebsocketprotocol-01
2	draft-ietf-hybi-thewebsocketprotocol-02
3	draft-ietf-hybi-thewebsocketprotocol-03
4	draft-ietf-hybi-thewebsocketprotocol-04
5	draft-ietf-hybi-thewebsocketprotocol-05
6	draft-ietf-hybi-thewebsocketprotocol-06
7	draft-ietf-hybi-thewebsocketprotocol-07
8	draft-ietf-hybi-thewebsocketprotocol-08
9	draft-ietf-hybi-thewebsocketprotocol-09

IANA is asked to add initial values to the registry, with suggested numerical values as these have been used in past versions of this protocol.

11.13. WebSocket Close Code Number Registry

This specification requests the creation of a new IANA registry for WebSocket Connection Close Code Numbers in accordance with the principles set out in RFC 5226 [RFC5226].
As part of this registry IANA will maintain the following information:

Status Code

The Status Code which denotes a reson for a WebSocket connection closure as per Section 7.4 of this document. The status code is an integer number.

Meaning

The meaning of the status code.

Contact

A contact for the entity reserving the status code.

Reference

The stable document requesting the status codes and defining their meaning, required for status codes in the range 1000-2999.

WebSocket Close Code Numbers are to be subject to different registration requirements depending on their range. Unless otherwise specified, requests are subject to Standards Action as per RFC5226 [RFC5226]. Requests for status codes for use by this protocol or subsequent versions are subject to Standards Action and should be granted status codes in the range 1000-1999. Requests for status codes for use by extensions are subject to Specification Required and should be granted Status Codes in the range 2000-2999. Requests for status codes for use by libraries and frameworks are subject to First Come First Served and should be granted in the range 3000-3999. The range of status codes from 4000-4999 is designated for Private Use by application code. Requests should indicate whether they are requesting status codes for use by the WebSocket protocol (or a future version of the protocol), by extensions, or by libraries and frameworks.

Status Code	Meaning	Contact	Reference
1000	Normal Closure	hybi@ietf.org	RFC XXXX
1001	Going Away	hybi@ietf.org	RFC XXXX
1002	Protocol error	hybi@ietf.org	RFC XXXX
1003	Unsupported Data	hybi@ietf.org	RFC XXXX
1004	Frame Too Large	hybi@ietf.org	RFC XXXX
1005	No Status Rcvd	hybi@ietf.org	RFC XXXX
1006	Abnormal Closure	hybi@ietf.org	RFC XXXX

IANA is asked to add initial values to the registry, with suggested numerical values as these have been used in past versions of this protocol.

11.14. WebSocket Opcode Registry

This specification requests the creation of a new IANA registry for WebSocket Opcodes in accordance with the principles set out in RFC 5226 [RFC5226].
As part of this registry IANA will maintain the following information:

Opcode

The opcode denotes the frame type of the WebSocket frame, as defined in Section 4.2. The status code is an integer number between 0 and 15, inclusive.

Meaning

The meaning of the opcode code.

Reference

The specification requesting the opcode.

WebSocket Opcode numbers are subject to Standards Action as per RFC5226 [RFC5226].

Opcode	Meaning	Reference
0	Continuation Frame	RFC XXXX
1	Text Frame	RFC XXXX
2	Binary Frame	RFC XXXX
8	Connection Close Frame	RFC XXXX
9	Ping Frame	RFC XXXX
10	Pong Frame	RFC XXXX

IANA is asked to add initial values to the registry, with suggested numerical values as these have been used in past versions of this protocol.

11.15. WebSocket Framing Header Bits Registry

This specification requests the creation of a new IANA registry for WebSocket Framing Header Bits in accordance with the principles set out in RFC 5226 [RFC5226]. This registry controls assignment of the bits marked RSV1, RSV2, and RSV3 in Section 4.2.
These bits are reserved for future versions or extensions of this specification.
WebSocket Framing Header Bits assignments are subject to Standards Action per RFC 5226 [RFC5226].

12. 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. [I-D.ietf-websec-origin]
- *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 URI using the steps to parse a WebSocket URI's components. These steps fail if the URI does not specify a WebSocket.
If at any time the connection is to be closed, then the specification needs to use the *Close the WebSocket Connection* algorithm (Section 7.1.1).
Section 7.1.4 defines when *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* (Section 6.2).
To send some data /data/ to an open connection, the specification needs to *Send a WebSocket Message* (Section 6.1).

13. 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 at http://whatwg.org/html5.
Special thanks also to John Tamplin for providing a significant amount of text for the Data Framing section of this specification.
Special thanks also to Adam Barth for providing a significant amount of text and background research for the Data Masking section of this specification.

14. References

14.1. Normative References

[ANSI.X3-4.1986]	American National Standards Institute, "Coded Character Set - 7-bit American Standard Code for Information Interchange", ANSI X3.4, 1986.
[FIPS. 180-2.2002]	National Institute of Standards and Technology, "Secure Hash Standard", FIPS PUB 180-2, August 2002.
[RFC1951]	Deutsch, P., "DEFLATE Compressed Data Format Specification version 1.3", RFC 1951, May 1996.
[RFC2119]	Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.
[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.
[RFC2817]	Khare, R. and S. Lawrence, "Upgrading to TLS Within HTTP/1.1", RFC 2817, May 2000.
[RFC2818]	Rescorla, E., "HTTP Over TLS", RFC 2818, May 2000.
[RFC3490]	Faltstrom, P., Hoffman, P. and A. Costello, "Internationalizing Domain Names in Applications (IDNA)", RFC 3490, March 2003.
[RFC3548]	Josefsson, S., "The Base16, Base32, and Base64 Data Encodings", RFC 3548, July 2003.
[RFC3629]	Yergeau, F., "UTF-8, a transformation format of ISO 10646", STD 63, RFC 3629, November 2003.
[RFC3864]	Klyne, G., Nottingham, M. and J. Mogul, "Registration Procedures for Message Header Fields", BCP 90, RFC 3864, September 2004.
[RFC3986]	Berners-Lee, T., Fielding, R. and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax", STD 66, RFC 3986, January 2005.
[RFC3987]	Duerst, M. and M. Suignard, "Internationalized Resource Identifiers (IRIs)", RFC 3987, January 2005.
[RFC4086]	Eastlake, D., Schiller, J. and S. Crocker, "Randomness Requirements for Security", BCP 106, RFC 4086, June 2005.
[RFC5246]	Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", RFC 5246, August 2008.
[RFC6066]	Eastlake, D., "Transport Layer Security (TLS) Extensions: Extension Definitions", RFC 6066, January 2011.

[RFC4648]	Josefsson, S., "The Base16, Base32, and Base64 Data Encodings", RFC 4648, October 2006.
[RFC5226]	Narten, T. and H. Alvestrand, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 5226, May 2008.
[RFC5234]	Crocker, D. and P. Overell, "Augmented BNF for Syntax Specifications: ABNF", STD 68, RFC 5234, January 2008.

14.2. Informative References

[WSAPI]	Hickson, I.E., "The Web Sockets API", August 2010.
[I-D.ietf-httpstate-cookie]	Barth, A, "HTTP State Management Mechanism", Internet-Draft draft-ietf-httpstate-cookie-20, December 2010.
[I-D.ietf-websec-origin]	Barth, A, "The Web Origin Concept", Internet-Draft draft-ietf-websec-origin-00, December 2010.
[RFC1950]	Deutsch, L.P. and J-L. Gailly, "ZLIB Compressed Data Format Specification version 3.3", RFC 1950, May 1996.
[RFC5321]	Klensin, J., "Simple Mail Transfer Protocol", RFC 5321, October 2008.
[RFC6202]	Loreto, S., Saint-Andre, P., Salsano, S. and G. Wilkins, "Known Issues and Best Practices for the Use of Long Polling and Streaming in Bidirectional HTTP", RFC 6202, April 2011.
[W3C.REC-wsc-ui-20100812]	Roessler, T. and A. Saldhana, "Web Security Context: User Interface Guidelines", World Wide Web Consortium Recommendation REC-wsc-ui-20100812, August 2010.

Author's Address

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