| | |
|---|---|
| HyBi Working Group | I.F. Fette |
| Internet-Draft | Google, Inc. |
| Intended status: Standards Track | February 09, 2011 |
| Expires: August 13, 2011 | |

The WebSocket protocol
draft-ietf-hybi-thewebsocketprotocol-05

## Abstract

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

## Status of this Memo



## Copyright Notice

## Table of Contents

## 1. Introduction

### 1.1. Background

*This section is non-normative.*
Historically, creating an instant messenger chat client as a Web application has required an abuse of HTTP to poll the server for updates while sending upstream notifications as distinct HTTP calls. This results in a variety of problems:

*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: 5
```

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-Nonce: AQIDBAUGBwgJCgsMDQ4PEC==
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 headers. The meaning of these headers is specified in Section 5 of this document. Additional headers may also be present, such as cookies 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. Broadly speaking, there are types for textual data, which is interpreted as UTF-8 text, binary data (whose interpretation is left up to the application), and control frames, which are not intended to

carry data for the application, but instead for protocol-level 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: 5
```

Headers in the handshake are sent by the client in a random order; the order is not meaningful.
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 "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.

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 abort the connection. 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.
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 version), 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, 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 must be treated as a failure and the websocket connection aborted. 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 WebSockets 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| constructor. 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. Upon receiving such a frame, the other peer sends an identical frame in acknowledgement, if it hasn't already sent one. Upon receiving *that* control frame, the first peer then closes the connection, safe in the knowledge that no further data is forthcoming.
After sending a control frame indicating the connection should be closed, a peer does not send any further data; after receiving a control frame indicating the connection should be closed, a peer discards any further data received.
It is safe for both peers to initiate this handshake simultaneously.
The closing handshake is intended to replace the TCP closing handshake (FIN/ACK), on the basis that the TCP closing handshake is not always reliable end-to-end, especially in the presence of man-in-the-middle proxies and other intermediaries.

## 1.5. Design philosophy

*This section is non-normative.*

The WebSocket protocol is designed on the principle that there should be minimal framing (the only framing that exists is to make the protocol frame-based instead of stream-based, and to support a distinction between Unicode text and binary frames). It is expected that metadata would be layered on top of WebSocket by the application layer, in the same way that metadata is layered on top of TCP by the application layer (HTTP).

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

The protocol is intended to be extensible; future versions will likely introduce additional concepts such as multiplexing and compression.

## 1.6. Security model

*This section is non-normative.*

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

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

It is similarly intended to fail to establish a connection when data from other protocols, especially HTTP, is sent to a WebSocket server, for example as might happen if an HTML |form| were submitted to a WebSocket server. This is primarily achieved by requiring that the server prove that it read the handshake, which it can only do if the handshake contains the appropriate parts which themselves can only be sent by a WebSocket handshake; in particular, fields starting with |Sec-| cannot be set by an attacker from a Web browser, even when using |XMLHttpRequest|.

### 1.7. Relationship to TCP and HTTP

*This section is non-normative.*
The WebSocket protocol is an independent TCP-based protocol. Its only relationship to HTTP is that its handshake is interpreted by HTTP servers as an Upgrade request.
Based on the expert recommendation of the IANA, the WebSocket protocol by default uses port 80 for regular WebSocket connections and port 443 for WebSocket connections tunneled over TLS.

### 1.8. Establishing a connection

*This section is non-normative.*
There are several options for establishing a WebSocket connection.
On the face of it, the simplest method would seem to be to use port 80 to get a direct connection to a WebSocket server. Port 80 traffic, however, will often be intercepted by HTTP proxies, which can lead to the connection failing to be established.
The most reliable method, therefore, is to use TLS encryption and port 443 to connect directly to a WebSocket server. This has the advantage of being more secure; however, TLS encryption can be computationally expensive.
When a connection is to be made to a port that is shared by an HTTP server (a situation that is quite likely to occur with traffic to ports 80 and 443), the connection will appear to the HTTP server to be a regular GET request with an Upgrade offer. In relatively simple setups with just one IP address and a single server for all traffic to a single hostname, this might allow a practical way for systems based on the WebSocket protocol to be deployed. In more elaborate setups (e.g. with load balancers and multiple servers), a dedicated set of hosts for WebSocket connections separate from the HTTP servers is probably easier to manage.

### 1.9. Subprotocols using the WebSocket protocol

*This section is non-normative.*
The client can request that the server use a specific subprotocol by including the |Sec-Websocket-Protocol| field in its handshake. If it is specified, the server needs to include the same field and one of the selected subprotocol values in its response for the connection to be established.
These subprotocol names do not need to be registered, but if a subprotocol is intended to be implemented by multiple independent WebSocket servers, potential clashes with the names of subprotocols defined independently can be avoided by using names that contain the domain name of the subprotocol's originator. For example, if Example Corporation were to create a Chat subprotocol to be implemented by many servers around the Web, they could name it "chat.example.com". If the Example Organization called their competing subprotocol

"example.org's chat protocol", then the two subprotocols could be implemented by servers simultaneously, with the server dynamically selecting which subprotocol to use based on the value sent by the client.
Subprotocols can be versioned in backwards-incompatible ways by changing the subprotocol name, 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. For readability, these words do not appear in all uppercase letters in this specification. [RFC2119]
Requirements phrased in the imperative as part of algorithms (such as "strip any leading space characters" or "return false and abort these steps") are to be interpreted with the meaning of the key word ("must", "should", "may", etc) used in introducing the algorithm.
Conformance requirements phrased as algorithms or specific steps may be implemented in any manner, so long as the end result is equivalent. (In particular, the algorithms defined in this specification are intended to be easy to follow, and not intended to be performant.)
Implementations may impose implementation-specific limits on otherwise unconstrained inputs, e.g. to prevent denial of service attacks, to guard against running out of memory, or to work around platform-specific limitations.
The conformance classes defined by this specification are user agents and servers.

### 2.1. Terminology

**ASCII** shall mean the character-encoding scheme defined in [ANSI.X3-4.1986].
**Converting a string to ASCII lowercase** means replacing all characters in the range U+0041 to U+005A (i.e. LATIN CAPITAL LETTER A to LATIN CAPITAL LETTER Z) with the corresponding characters in the range U+0061 to U+007A (i.e. LATIN SMALL LETTER A to LATIN SMALL LETTER Z).
Comparing two strings in an **ASCII case-insensitive** manner means comparing them exactly, code point for code point, except that the characters in the range U+0041 to U+005A (i.e. LATIN CAPITAL LETTER A to LATIN CAPITAL LETTER Z) and the corresponding characters in the

range U+0061 to U+007A (i.e. LATIN SMALL LETTER A to LATIN SMALL LETTER Z) are considered to also match.
The term "URL" is used in this section in a manner consistent with the terminology used in HTML, namely, to denote a string that might or might not be a valid URI or IRI and to which certain error handling behaviors will be applied when the string is parsed. [HTML]
When an implementation is required to *send* data as part of the WebSocket protocol, the implementation may delay the actual transmission arbitrarily, e.g. buffering data so as to send fewer IP packets.

## 3. WebSocket URLs

### 3.1. Parsing WebSocket URLs

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

1. If the /url/ string is not an absolute URL, then fail this algorithm. [RFC3986] [RFC3987]

2. Resolve the /url/ string using the resolve a Web address algorithm defined by the Web addresses specification, with the URL character encoding set to UTF-8. [RFC3629] [RFC3986] [RFC3987]

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

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

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

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

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

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

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

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

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

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

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

### 3.2. Constructing WebSocket URLs

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

1. Let /url/ be the empty string.

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

3. Append /host/ to /url/.

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

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

6. Return /url/.

### 3.3. Valid WebSocket URLs

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

*The /host/ must be ASCII-only (i.e. it must have been punycode-encoded already if necessary, and MUST NOT contain any characters above U+007E).

*The /resource name/ string must be a non-empty string of characters in the range U+0021 to U+007E that starts with a U+002F SOLIDUS character (/).

Any WebSocket URLs not meeting the above criteria are considered invalid, and a client MUST NOT attempt to make a connection to an invalid WebSocket URL. A client SHOULD attempt to parse a URL obtained from any external source (such as a web site or a user) using the steps specified in Section 3.1 to obtain a valid WebSocket URL, but MUST NOT

attempt to connect with such an unparsed URL, and instead only use the parsed version and only if that version is considered valid by the criteria above.

## 4. Data Framing

### 4.1. Overview

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. As such, In the absence of extensions negotiated during the opening handshake (Section 5), all reserved bits MUST be 0 and reserved opcode values MUST NOT be used.
A data frame MAY be transmitted by either the client or the server at any time after handshake completion and before that host has generated a close message (Section 4.5.1).

### 4.2. Client-to-Server Masking

The client MUST mask all frames sent to the server.
The masking-key is contained completely within the frame.
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.

```
masked-frame  = masking-key masked-data
masking-key = 4full-octet
masked-data   = *full-octet
full-octet    = %x00-FF
```

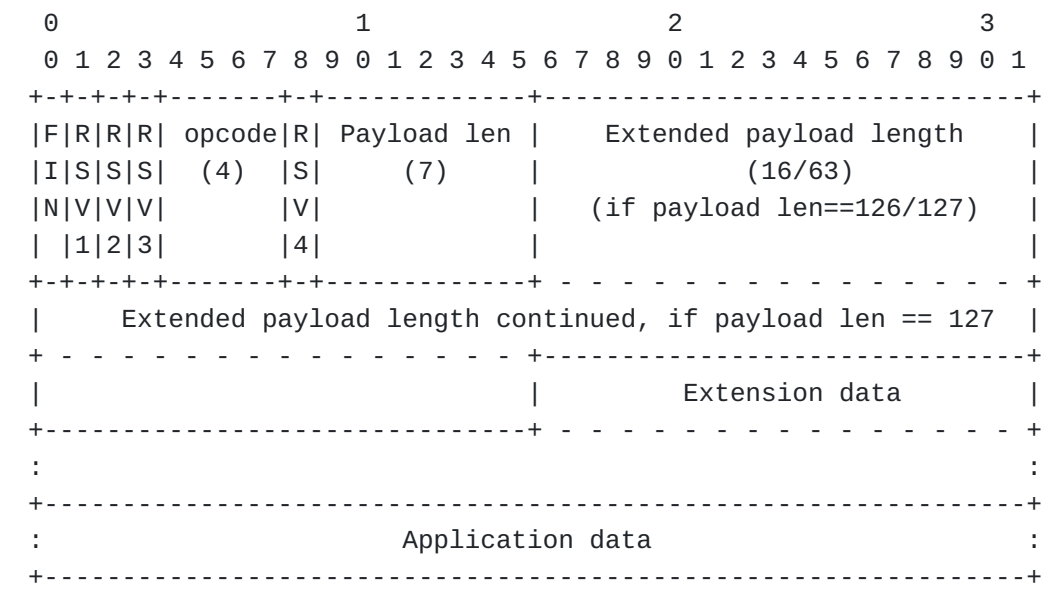
Each masked frame consists of a 32-bit masking-key followed by masked-data:
The masked-data is the clear-text frame "encrypted" using a simple XOR cipher as follows.
Octet i of the masked-data is the XOR of octet i of the clear text frame with octet i modulo 4 of the masking-key:

```
j              = i MOD 4
masked-octet-i = clear-text-octet-i XOR octet-j-of-masking-key
```

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-nonce is essential to prevent the author of malicious application data from selecting the bytes that appear on the wire.

### 4.3. Base Framing Protocol

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




**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, RSV4:** 1 bit each

Must be 0 unless an extension is negotiated which defines meanings for non-zero values

**Opcode:** 4 bits

Defines the interpretation of the payload data

**Payload length:** 7 bits

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

a 64-bit unsigned integer (the high bit must be 0) are the payload length. 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.

**Extension data:** n bytes

The extension data is 0 bytes unless there is a reserved op-code or reserved bit present in the frame which indicates an extension has been negotiated. Any extension MUST specify the length of the extension data, or how that length may be calculated, and its use MUST be negotiated during the handshake. If present, the extension data is included in the total payload length.

**Application data:** n bytes

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

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-rsv4
                          frame-length
                          frame-extension
                          application-data;

frame-fin               = %x0 ; more frames of this message follow
                        / %x1 ; final frame of 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 ; connection close
                        / %x2 ; ping
                        / %x3 ; pong
                        / %x4 ; text frame
                        / %x5 ; binary frame
                        / %x6-F ; reserved

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

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

frame-length-16         = %x0000-FFFF

frame-length-63         = %x0000000000000000-7FFFFFFFFFFFFFFF

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

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

### 4.4. Fragmentation

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. Its content is the concatenation of the application data from each of those frames in order.

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

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

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

*An intermediary MAY change the fragmentation of a message if the message uses only opcode and reserved bit values known to the intermediary.

### 4.5. Control Frames

Control frames have opcodes of 0x01 (Close), 0x02 (Ping), or 0x03 (Pong). Control frames are used to communicate state about the websocket.
All control frames MUST be 125 bytes or less in length and MUST NOT be fragmented.

#### 4.5.1. Close

The Close message contains an opcode of 0x01.
The Close message contains a body that is at least one byte in length. If the close is initiated by the client, the first byte of the body MUST be a 0x43. If the close is initiated by the server, the first byte of the body MUST be a 0x53. The body MAY contain additional bytes, the meaning of those bytes are not defined by this version of the protocol.
The application MUST NOT send any more data messages after sending a close message.
A received close message is deemed to be an acknowledgement if the message body matches the body of a close message previously sent by the receiver.
Upon receipt of an initiated close the endpoint MUST send a close acknowledgment. It should do so as soon as is practical. The body of the acknowledgement must match the body of the close message received.
The websocket is considered fully closed when an endpoint has either received a close acknowledgment or sent a close acknowledgment.
To prevent a situation where ambiguity arises from a client and server both sending a close frame at approximately the same time, the close frame has a body that can be used to distinguish whether a close frame is an initiation of a close event or the acknowledgement of the other

side's close. (Without resolving the ambiguity, a party has no way of knowing whether its last bytes before the close frame were received).

### 4.5.2. Ping

The Ping message contains an opcode of 0x02.
Upon receipt of a Ping message, an endpoint MUST send a Pong message in response. It SHOULD do so as soon as is practical. The message bodies of the Ping and Pong MUST be the same.

### 4.5.3. Pong

The Pong message contains an opcode of 0x03.
Upon receipt of a Ping message, an endpoint MUST send a Pong message in response. It SHOULD do so as soon as is practical. The message bodies of the Ping and Pong MUST be the same. A Pong is issued only in response to the most recent Ping.

## 4.6. Data Frames

All frame types not listed in Section 4.5 are data frames, which transport application-layer data. The opcode determines the interpretation of the application data:

**Text**

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

**Binary**

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

## 4.7. Examples

*This section is non-normative.*

*A single-frame text message

-0x84 0x05 0x48 0x65 0x6c 0x6c 0x6f (contains "Hello")

*A fragmented text message

-0x04 0x03 0x48 0x65 0x6c (contains "Hel")

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

*Ping request and response

-0x82 0x05 0x48 0x65 0x6c 0x6c 0x6f (contains a body of "Hello", but the contents of the body are arbitrary)

-0x83 0x05 0x48 0x65 0x6c 0x6c 0x6f (contains a body of "Hello", matching the body of the ping)

*256 bytes binary message in a single frame

-0x85 0x7E 0x0100 [256 bytes of binary data]

*64KiB binary message in a single frame

-0x85 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 handshake. This specification provides opcodes 0x6 through 0xF, the extension data field, and the frame-rsv1, frame-rsv2, frame-rsv3, and frame-rsv4 bits of the frame header for use by extensions. The negotiation of extensions is discussed in further detail in Section 8.1. Below are some anticipated uses of extensions. This list is neither complete nor proscriptive.

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

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

*Reserved opcode values can be defined.

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

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

## 5. Opening Handshake

### 5.1. Client Requirements

User agents 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 user agent 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 user agent is to **establish a WebSocket connection** to a WebSocket URL /url/, it must meet the following requirements. In the following text, we will use terms from Section 3 such as "/host/" and "/secure/ flag" as defined in that section.

1. The WebSocket URL and its components MUST be valid according to Section 3.3. If any of the requirements are not met, the client MUST fail the WebSocket connection and abort these steps.

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

   If the user agent 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 user agent MUST assume for the purposes of this step that each host name refers to a distinct remote host, but should instead limit the total number of simultaneous connections that are not established to a reasonably low number (e.g., in a Web browser, to 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 user agent can have with a single remote host. Servers can refuse to connect users with an excessive number of connections, or disconnect resource-hogging users when suffering high load.

3. *Proxy Usage*: If the user agent is configured to use a proxy when using the WebSocket protocol to connect to host /host/ and/or port /port/, then the user agent 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 user agent uses an HTTP proxy for all traffic, then if it was to try to connect to port 80 on server example.com, it might send the following lines to the proxy server:

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

If the user agent 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 URL to pass the function must be constructed from /host/, /port/, /resource name/, and the /secure/ flag using the steps to construct a WebSocket URL.

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

4. If the connection could not be opened, either because a direct connection failed or because any proxy used returned an error, then the user agent MUST fail the WebSocket connection and abort the connection attempt.

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

   User agents MUST use the Server Name Indication extension in the TLS handshake. [RFC4366]

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 a 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 URL 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.

4. The request MUST contain a "Host" header whose value is equal to the authority component of the WebSocket URL.

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

6. The request MUST contain a "Connection" header whose value is equal to "Upgrade".

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 randomly 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].

   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 5.

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. The elements that comprise this value MUST be non-empty strings with characters in the range U+0021 to U+007E and MUST all be unique. The ABNF for the value of this header is 1#(token | quoted-string), 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 ABNF for the value of this header is 1#(token | quoted-string), where the definitions of constructs and rules are as given in [RFC2616]. The interpretation of this header is described in Section 8.1.

12. The request MAY include headers associated with sending cookies, as defined by the appropriate specifications [I-D.ietf-httpstate-cookie].

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:

*If the status code received from the server is not 101, the client MUST fail the WebSocket connection.

*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.

*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.

*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.

Where the algorithm above requires that a user agent fail the WebSocket connection, the user agent may first read an arbitrary number of further bytes from the connection (and then discard them) before actually **failing the WebSocket connection**. Similarly, if a user agent can show that the bytes read from the connection so far are such that

there is no subsequent sequence of bytes that the server can send that would not result in the user agent being required to **fail the WebSocket connection**, the user agent may immediately **fail the WebSocket connection** without waiting for those bytes.
NOTE: The previous paragraph is intended to make it conforming for user agents to implement the algorithm in subtly different ways that are equivalent in all ways except that they terminate the connection at earlier or later points. For example, it enables an implementation to buffer the entire handshake response before checking it, or to verify each field as it is received rather than collecting all the fields and then checking them as a block.

### 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 Web Socket 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 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 abort the WebSocket connection.

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-Origin" header.

5. A "Sec-WebSocket-Version" header, with a value of 5.

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 8.1.

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

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 handshake) must run through the encrypted tunnel. [RFC2246]

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.

   **/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 connection. The server MAY

send a non-200 response code with a |Sec-WebSocket-Version| header indicating the version(s) the server is capable of understanding along with this non-200 response code.

**/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.

**/subprotocol/**
A (possibly empty) list representing the subprotocol the server is ready to use. If the server supports multiple subprotocols, then the value should be derived from the client's handshake, specifically 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.

**/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 should 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 8.1.

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

    1. A 101 response code. Such a response could look like "HTTP/1.1 101 Switching Protocols"

    2. 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.

        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.

3. A "Sec-WebSocket-Nonce" header. 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 randomly 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==" value

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

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

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

## 6. Error Handling

### 6.1. Handling errors in UTF-8 from the server

When a client is to interpret a byte stream as UTF-8 but finds that the byte stream is not in fact a valid UTF-8 stream, then any bytes or sequences of bytes that are not valid UTF-8 sequences must be interpreted as a U+FFFD REPLACEMENT CHARACTER.

### 6.2. Handling errors in UTF-8 from the client

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

## 7. Closing the connection

### 7.1. Abnormal closures

#### 7.1.1. Client-initiated closure

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

*A server whose host name could not be resolved.

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

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

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

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

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

#### 7.1.2. Server-initiated closure

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

### 7.2. Normal closure of connections

To **close the WebSocket connection**, the user agent MUST first send a Close control frame, as described in Section 4.5.1. Upon receiving a Close control frame, the other party sends a matching Close control frame. Once an endpoint has sent or received an acknowledgement of a Close control frame, that endpoint must close the TCP connection, using whatever mechanism possible (e.g. either the TCP RST or FIN mechanisms). Whether the user agent or the server closes the connection first, it is said that the **WebSocket connection is closed**. If the connection was closed after the client finished the WebSocket closing

handshake, then the WebSocket connection is said to have been closed *cleanly*.
Servers may close the WebSocket connection whenever desired. User agents should not close the WebSocket connection arbitrarily.

## 8. 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.

### 8.1. 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:

```
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 | quoted-string ) ]
```

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 accepts 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. 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.

### 8.2. 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.

#### 8.2.1. Compression

The registered extension token for this compression extension is "deflate-stream".
The extension does not have any per message extension data and it does not define the use of any WebSocket reserved bits or op codes.
Senders using this extension MUST apply RFC 1951 encodings to all bytes of the data stream following the handshake including both data and control messages. The data stream MAY include multiple blocks of both compressed and uncompressed types as defined by RFC 1951. [RFC1951]
Senders MUST NOT delay the transmission of any portion of a WebSocket message because the deflate encoding of the message does not end on a byte boundary. The encodings for adjacent messages MAY appear in the same byte if no delay in transmission is occurred by doing so.

## 9. Security considerations

While this protocol is intended to be used by scripts in Web pages, it can also be used directly by hosts. Such hosts are acting on their own behalf, and can therefore send fake "Origin" fields, misleading the server. Servers should therefore be careful about assuming that they are talking directly to scripts from known origins, and must consider that they might be accessed in unexpected ways. In particular, a server should not trust that any input is valid.
EXAMPLE: For example, if the server uses input as part of SQL queries, all input text should be escaped before being passed to the SQL server, lest the server be susceptible to SQL injection.

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

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

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

## 10. IANA considerations

### 10.1. Registration of ws: scheme

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

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

**URI scheme name.**

ws

**Status.**
Permanent.

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

**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.

**Interoperability considerations.**
None.

**Security considerations.**
See "Security considerations" section above.

**Contact.**
Ian Hickson <ian@hixie.ch>

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

**References.**

This document.

### 10.2. Registration of wss: scheme

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

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

**URI scheme name.**

wss

**Status.**

Permanent.

**URI scheme syntax.**

In ABNF terms using the terminals from the URI specifications: [RFC5234][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.**
          Ian Hickson <ian@hixie.ch>

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

**References.**
             This document.

### 10.3. Registration of the "WebSocket" HTTP Upgrade keyword

**Name of token.**
                WebSocket

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

**Contact.**
          Ian Hickson <ian@hixie.ch>

**References.**
             This document.

### 10.4. Sec-WebSocket-Key and Sec-WebSocket-Nonce

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

**Header field name**
                  Sec-WebSocket-Key

**Applicable protocol**
                    http

**Status**
        reserved; do not use outside WebSocket handshake

**Author/Change controller**
                         IETF

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

**Related information**
                    None.

**Header field name**
                  Sec-WebSocket-Nonce

**Applicable protocol**

http

**Status**

reserved; do not use outside WebSocket handshake

**Author/Change controller**

IETF

**Specification document(s)**

This document is the relevant specification.

**Related information**

None.

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

### 10.5. 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**

reserved; do not use outside WebSocket handshake

**Author/Change controller**

IETF

**Specification document(s)**

This document is the relevant specification.

**Related information**

None.

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

### 10.6. 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**
reserved; do not use outside WebSocket handshake

**Author/Change controller**
IETF

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

**Related information**
None.

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

### 10.7. 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**
reserved; do not use outside WebSocket handshake

**Author/Change controller**
IETF

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

**Related information**
None.

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

### 10.8. 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**
reserved; do not use outside WebSocket handshake

**Author/Change controller**
IETF

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

**Related information**
None.

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

### 10.9. 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**
reserved; do not use outside WebSocket handshake

**Author/Change controller**
IETF

**Specification document(s)**

This document is the relevant specification.

**Related information**

None.

The |Sec-WebSocket-Version| header is used in the WebSocket 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 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. 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 URL using the steps to parse a WebSocket URL's components. These steps fail if the URL does not specify a WebSocket.
If a connection can be established, then it is said that the "WebSocket connection is established".
If at any time the connection is to be closed, then the specification needs to use the "close the WebSocket connection" algorithm.
When the connection is closed, for any reason including failure to establish the connection in the first place, it is said that the "WebSocket connection is closed".
While a connection is open, the specification will need to handle the cases when "a WebSocket message has been received" with text /data/.
To send some text /data/ to an open connection, the specification needs to "send /data/ using the WebSocket".

## 12. Acknowledgements

Special thanks are due to Ian Hickson, who was the original author and editor of this protocol. The initial design of this specification benefitted from the participation of many people in the WHATWG and WHATWG mailing list. Contributions to that specification are not tracked by section, but a list of all who contributed to that specification is given in the WHATWG HTML specification. [HTML]
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.

## 13. References

| | |
|---|---|
| **[HTML]** | Hickson, I.E., "HTML", August 2010. |
| **[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. |
| **[RFC2246]** | Dierks, T. and C. Allen, "The TLS Protocol Version 1.0", RFC 2246, January 1999. |
| **[RFC2616]** | Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P. and T. Berners-Lee, "Hypertext Transfer Protocol -- HTTP/1.1", RFC 2616, June 1999. |
| **[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. |
| **[RFC4366]** | Blake-Wilson, S., Nystrom, M., Hopwood, D., Mikkelsen, J. and T. Wright, "Transport Layer Security (TLS) Extensions", RFC 4366, April 2006. |
| **[RFC5234]** | Crocker, D. and P. Overell, "Augmented BNF for Syntax Specifications: ABNF", STD 68, RFC 5234, January 2008. |
| **[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. |
| **[WSAPI]** | Hickson, I.E., "The Web Sockets API", August 2010. |


## Author's Address


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