HyBi Working Group   I. Fette
Internet-Draft   Google, Inc.
Intended status: Standards Track   February 25, 2011
Expires: August 29, 2011

The WebSocket protocol
draft-ietf-hybi-thewebsocketprotocol-06

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:

- o The server is forced to use a number of different underlying TCP connections for each client: one for sending information to the client, and a new one for each incoming message.
- o The wire protocol has a high overhead, with each client-to-server message having an HTTP header.
- o The client-side script is forced to maintain a mapping from the outgoing connections to the incoming connection to track replies.

A simpler solution would be to use a single TCP connection for traffic in both directions. This is what the 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: 6
```

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 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: 6
```

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 if semantics of that status code are not defined in the context of a

WebSocket connection, 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 (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 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.

By sending a close frame and waiting for a close frame in response,

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.

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, 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.

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.

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 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 "URI" 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. [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

3.1. Parsing WebSocket URIs

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

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

2. Resolve the /uri/ string using the resolve a Web address algorithm defined by the Web addresses specification, with the URI 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 URI at this point.

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

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

5. If the <scheme> component of /uri/ is "ws", set /secure/ to false; otherwise, if the <scheme> component is "wss", set /secure/ to true; otherwise, fail this algorithm.

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

7. If /uri/ 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 /uri/.

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

11. If /uri/ 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 URIs

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

1. Let /uri/ be the empty string.

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

3. Append /host/ to /uri/.

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 /uri/.

5. Append /resource name/ to /uri/.

6. Return /uri/.

3.3. Valid WebSocket URIs

For a WebSocket URI 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 URIs not meeting the above criteria are considered invalid, and a client MUST NOT attempt to make a connection to an invalid WebSocket URI. A client SHOULD attempt to parse a URI 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 URI, but MUST NOT attempt to connect with such an unparsed URI, 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 endpoint has sent 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.

Each masked frame consists of a 32-bit masking-key followed by masked-data:

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

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]

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-------+-+-------------+-------------------------------+
|F|R|R|R| opcode|R| Payload len |    Extended payload length    |
|I|S|S|S|  (4)  |S|     (7)     |             (16/63)           |
|N|V|V|V|       |V|             |   (if payload len==126/127)   |
| |1|2|3|       |4|             |                               |
+-+-+-+-+-------+-+-------------+ - - - - - - - - - - - - - - - +
|     Extended payload length continued, if payload len == 127  |
+ - - - - - - - - - - - - - - - +-------------------------------+
|                               |         Extension data        |
+-------------------------------+ - - - - - - - - - - - - - - - +
:                                                               :
+---------------------------------------------------------------+
:                       Application data                        :
+---------------------------------------------------------------+
```

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 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:

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

- o 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. As an example, for a text message sent as three fragments, the first fragment would have an opcode of 0x4 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.

- o Control frames MAY be injected in the middle of a fragmented message. Control frames themselves MUST NOT be fragmented. _Note: if control frames could not be interjected, the latency of a ping, for example, would be very long if behind a large message. As such, an endpoint MUST be capable of handling control frames in the middle of a fragmented message._

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

- o Clients and servers MUST support receiving both fragmented and unfragmented messages.

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

- o 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.

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. Control frames can be interjected in the middle of a fragmented message.

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 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 message too large, or an endpoint having received a message 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 integer (in network byte order) representing a status code defined in Section 7.4. Following the 2-byte integer the body MAY contain UTF-8 encoded data, the interpretation of which is not defined by this specification.

The application MUST NOT send any more data messages after sending a close message.

If an endpoint receives a Close message and that endpoint did not previously send a Close message, the endpoint MUST send a Close message in response. It SHOULD do so as soon as is practical.

After both sending and receiving a close message, an endpoint considers the websocket connection closed, and SHOULD close the underlying TCP connection.

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 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.

- o A single-frame text message
  - * 0x84 0x05 0x48 0x65 0x6c 0x6c 0x6f (contains "Hello")
- o A fragmented text message
  - * 0x04 0x03 0x48 0x65 0x6c (contains "Hel")
  - * 0x80 0x02 0x6c 0x6f (contains "lo")
- o 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.

When the user agent is to *establish a WebSocket connection* to a WebSocket URI /uri/, 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 URI 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. There MUST be no more than one connection in a CONNECTING state. 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:

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

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

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

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

   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 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.

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

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 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 6.

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 interpretation and format 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:

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

o 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 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 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-Version" header, with a value of 6.

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 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 Paragraph 2 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. Optionally, a "Sec-WebSocket-Protocol" header, with a value /subprotocol/ as defined in Paragraph 2 of Section 5.2.2.

    4. Optionally, a "Sec-WebSocket-Extensions" header, with a value /extensions/ as defined in Paragraph 2 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. 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, discarding any trailing bytes that may be received. And endpoint MAY close the connection via any means available when necessary, such as when under attack.

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_, and endpoint MUST send a Close control frame, as described in Section 4.5.1. Upon receiving a Close control frame, the other party sends a Close control frame in response. 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 Connection Is Closed

When the underlying TCP connection is closed, it is said that _the WebSocket connection is closed_. If the tcp connection was closed after the WebSocket closing handshake was completed, the WebSocket connection is said to have been closed _cleanly_.

7.1.4. Fail the WebSocket Connection

Certain algorithms and specifications require a 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) in an appropriate manner.

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.

7.2. Abnormal closures

7.2.1. Client-initiated closure

Certain algorithms, namely during the initial handshake, require the user agent to *fail the WebSocket connection*. To do so, the user agent must _Close the WebSocket connection_ as previously defined, and may report the problem to the user via an appropriate mechanism (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.

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. User agents 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 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 message that is too large.

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 interpretaion of these codes is undefined by this protocol.

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-strea
m
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.

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 message from a client as a request, and a message 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 message that looks to an intermediary like a GET request for a common piece of JavaScript on another domain, and send back a message that is interpreted as a cacheable response to that request, thus poisioning the cache for other users. To prevent this attack, messages 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 message that can be misinterpreted by a proxy as a non-WebSocket request.

10. IANA considerations

10.1. Registration of ws: scheme

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]

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

The path and query components form the resource name sent to the server to identify the kind of service desired. Other components have the meanings described in RFC3986.

URI scheme semantics.
The only operation for this scheme is to open a connection using the WebSocket protocol.

Encoding considerations.
Characters in the host component that are excluded by the syntax defined above must be converted from Unicode to ASCII by applying the IDNA ToASCII algorithm to the Unicode host name, with both the AllowUnassigned and UseSTD3ASCIIRules flags set, and using the result of this algorithm as the host in the URI. [RFC3490]

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

A |wss:| URI 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]

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

The path and query components form the resource name sent to the server to identify the kind of service desired. Other components have the meanings described in RFC3986.

URI scheme semantics.
The only operation for this scheme is to open a connection using the WebSocket protocol, encrypted using TLS.

Encoding considerations.
Characters in the host component that are excluded by the syntax defined above must be converted from Unicode to ASCII by applying the IDNA ToASCII algorithm to the Unicode host name, with both the AllowUnassigned and UseSTD3ASCIIRules flags set, and using the result of this algorithm as the host in the URI. [RFC3490]

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

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
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| header is used in the WebSocket 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 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.

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:

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

13. Appendix: List of Changes

This section is not normative. This section was added at the request of the chairs to help track changes between versions. This section will be removed from the final version of this document.

13.1. Changes from -05 to -06

Two major areas were changed in this draft. The closing handshake was clarified and re-written to add in terminology matching the API specification. The close frame was given an optional status code to indicate closure reason, and the notion of a body indicating which side initiated the close removed. Aside from this, many areas were clarified in areas previously ambiguous, though the meaning should remain consistent with the intent of previous drafts. Certain other material changes that are not as large as those previously mentioned are listed below, though for a complete list readers are reminded that a tool is available to diff two versions at http://tools.ietf.org/tools/rfcdiff/. The list below is my attempt at a changelog, not an authoritative guarantee, plese use the diff tool for a complete list.

- Clarified that Sec-WebSocket-Origin is optional for non-browser clients.

- Clarified the semantics of the closing handshake to be that the connection is closed when an endpoint has both sent and received a close frame.

- Changed text around final HTTP responses and the WebSocket handshake.

- Removed Sec-WebSocket-Nonce

- Attempted to convert use of URL to URI terminology. (Ticket 41)

- Attempted to resolve Ticket 42 re: HTML spec reference.

- Edited potentially misleadin text around the word "even" in Section 1.6 and what applied to XHR vs more broadly.

- Removed non-material text from 1.8 about establishing a connection.

- Clarified text in the section about fragmentation (4.4). No material changes, clarification only.

o Clarified that control frames (4.5) may be interjected in the middle of a fragmented message.

o Clarified what was meant by the body of a close frame.

o Clarified the intent in 5.1 that there be only one connection in CONNECTING state.

o Cleaned 1.5 up to note that compression was already introduced in the spec, left in multiplexing as a future definition.

o Randomly selected randomly - typo fix.

o Added a change log in the appendix.

o Included in security considerations a description of the attack presented by Adam Barth.

o Changed some referneces from Web-Socket to WebSocket

o Clarified in 3.1 that only ws and wss are valid options, and that other schemes should result in a failure.

o Various cleanups around terminology of "host", "endpoint", and "user agent".

o Defined status codes and reserved ranges for close frames.

o Added text that a TCP connection should be shut down cleanly.

o Clarified whether the upgrade header exactly equaled upgrade or contained an upgrade token.

14. 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, <http://csrc.nist.gov/publications/fips/fips180-2/fips180-2.pdf>.

[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", draft-ietf-httpstate-cookie-20 (work in progress), December 2010.

[I-D.ietf-websec-origin]
Barth, A., "The Web Origin Concept", draft-ietf-websec-origin-00 (work in progress), December 2010.

[WSAPI] Hickson, I., "The Web Sockets API", August 2010, <http://dev.w3.org/html5/websockets/>.

Author's Address

Ian Fette
Google, Inc.

Email: ifette+ietf@google.com
URI: http://www.ianfette.com/

HYBI Working Group
Internet-Draft
Intended status: Informational
Expires: September 15, 2011

G. Montenegro, Ed.
Microsoft Corporation

March 14, 2011

HyBi WebSocket Requirements and Features
draft-ietf-hybi-websocket-requirements-02

Abstract

This document states the requirements of the WebSocket Protocol. The goal of the document is to provide a stable base for protocol design and related discussion.

Status of this Memo

Copyright Notice

Table of Contents

1. Introduction

HTTP [RFC2616] is a client/server protocol, where the HTTP servers store the data and provide it when it is requested by clients. When used to retrieve data from an HTTP server, the client sends HTTP requests to the server, and the server returns the requested data in HTTP responses. So the client has to poll the server continuously in order to receive new data.

Recently, techniques that enable bidirectional communication over HTTP have become more pervasive. Those techniques reduce the need to poll continuously the server thanks to the usage of HTTP hanging requests and multiple connections between the client and the server [I-D.ietf-hybi-design-space].

The goal of HyBi is to provide an efficient and clean two-way communication channel between client and server.

The communication channel will:

- Allow each side to, independently from the other, send data when it is willing and ready to do so.
- Rely on a single TCP connection for traffic in both directions.
- Reduce the high overhead produced by HTTP headers in each request/ response.

The goal of this work is to provide the set of requirements for the WebSocket Protocol.

In the following sections we list and analyse the requirements from the perspective of clients and servers.

2. Terminology

This document uses the following HyBi-related terms:

connection: A transport layer virtual circuit established between a client and a server for the purpose of communication.

frame: The basic unit of WebSocket communication, consisting of a structured sequence of octets matching the syntax defined in the actual protocol and transmitted on the established communication channel.

message: user message: a block of related data with identified boundaries. A message may comprise multiple frames.

origin server: The server on which a given resource resides or is to be created.

WebSocket handshake: The process (and associated capability negotiation) that sets up the WebSocket communication channel.

WebSocket communication channel: After the WebSocket handshake is complete, the resultant bi-directional communication path between client and server over the transport (e.g., TCP, or SSL over TCP).

WebSocket sub-protocol: The negotiated sub-protocol for use on a WebSocket communication channel that dictates framing, encoding, etc.

2.1. Requirements Language

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 [RFC2119].

3. WebSocket Requirements

The following requirements for the WebSocket Protocol have been identified both in the HyBi wg input document [I-D.ietf-hybi-thewebsocketprotocol] and in the HyBi mailing list dicussion.

REQ. 1: The WebSocket Protocol MUST run directly on top of the transport protocol (over which the communication was running up to and including the WebSocket handshake). The transport protocol is limited to TCP.

REQ. 2: The WebSocket Protocol MUST be able to handle (send and receive) messages on the transport protocol (over which the communication was running up to and including the WebSocket handshake).

Reason: transfer data as message obviates the need for the receiver to parse/handle partial content.

REQ. 3: The protocol MUST support the ability to fragment a message into frames of a given length.

REQ. 4: It MUST be possible to send a message when the total size is either unknown or exceeds a fixed buffer size.

Reason: This will allow dynamic messages to be constructed and sent without the need to buffer the entire message.

REQ. 5: Textual data MUST be encoded as UTF-8.

REQ. 6: The protocol MUST support and clearly distinguish between textual and binary data types (e.g. binary) via a common framing with explicit length indication.

REQ. 7: The WebSocket protocol MUST allow HTTP and WebSocket connections to be served from the same port. Consideration MUST be given:

* to provide WebSocket services via modules that plug in to existing web infrastructure.

* to making it possible and practical to implement standalone implementations of the protocol without requiring a fully conforming HTTP implementation.

Reason: Some server developers would like to integrate WebSocket support into existing HTTP servers. In addition, the default HTTP and HTTPS ports are ofter favoured for traffic that has to go through a firewall, so service providers will likely want to be able to use WebSocket over ports 80 and 443, even when running a Web server on the same host. However there could be scenarios where it is not opportune or possible to setup a proxy on the same HTTP server.

REQ. 8: If using an HTTP Upgrade exchange in the WebSocket handshake, the protocol MUST be HTTP compatible up to and including the Upgrade exchange.

REQ. 9: The protocol SHOULD make it possible and practical to reuse existing HTTP components where appropriate.

Reason: Reusing existing well-debugged software decreases the number of implementation errors as well as the possibility to introduce security holes, and increases development speed, especially when the WebSocket server is implemented as modules that plug in to existing popular Web servers.

3.1. WebSocket Client Requirements

REQ. 10: The WebSocket Client MUST be able to set up a communication channel with a WebSocket Server using a well-defined handshake.

REQ. 11: The WebSocket Protocol MUST provide for graceful close of an active WebSocket connection on request from the user Application.

Reason: a clean shutdown signals that the other endpoint has definitely received all the messages sent prior to the close, so there is no protocol uncertainty about what has been processed and what can be retried on another connection.

REQ. 12: WebSocket Protocol MUST also allow ungraceful close, either on request from the user application or as a result of a detected error condition.

REQ. 13: The WebSocket Client MUST be able to request the server, during the handshake, to use a specific WebSocket sub-protocol.

REQ. 14: The WebSocket Client MUST have the ability to send and clearly distinguish between arbitrary text or binary content to the server on the established communication channel.

3.2. WebSocket Server Requirements

REQ. 15: The WebSocket Server that accepts to set up a communication channel with a WebSocket Client MUST use a well-defined handshake.

REQ. 16: The WebSocket Server MUST have the ability to send and clearly distinguish between arbitrary text or binary content to the client on the established communication channel.

3.3. WebSocket Proxies Requirements

Todo

REQ. 17: The WebSocket protocol MUST work over existing proxies to the same extent as HTTP or HTTPS already does.

Reason: This is in line with Req on HTTP compliance.

3.4. WebSocket Security Requirements

REQ. 18: The WebSocket Protocol MUST use the Origin-based security model commonly used by Web browsers to restrict which Web pages can contact a WebSocket sever when the WebSocket protocol is used from a Web page.

REQ. 19: When used directly (not from a Web page), the WebSocket Protocol MUST use a security model equivalent to that of direct HTTP or HTTPS usage.

REQ. 20: WebSocket should be designed to be robust against cross-protocol attacks. The protocol design should consider and mitigate the risk presented by WebSocket clients to existing servers (including HTTP servers). It should also consider and mitigate the risk to WebSocket servers presented by clients for other protocols (including HTTP).

Reason: As the WebSocket protocol is expected to be often used in browsers, a careful design is necessary to mitigate the chances for hostile JavaScript to use WebSocket for a cross-protocol attack against vanilla HTTP resources or non-HTTP servers. More the design should prevent the possibility for cross-site XMLHttpRequest (using CORS or XDomainRequest) to be used for a cross-protocol attack against WebSocket resources, potentially violating integrity (though not confidentiality).

Subsequent discussion in the working group has determined consensus on the use of masking as one of the mechanisms to mitigate this concern.

4. Security Considerations

5. IANA Considerations

This requirements document does not mandate any immediate IANA actions. However, such IANA considerations may arise from future HyBi specification documents which try to meet the requirements given here.

6. Acknowledgments

The initial requirements were created by Salvatore Loreto. Thanks to Greg Wilkins and Maciej Stachowiak for fulfilling previous editing duties.

7. References

7.1. Normative References

[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.

[I-D.ietf-hybi-thewebsocketprotocol] Fette, I., "The WebSocket protocol", draft-ietf-hybi-thewebsocketprotocol-06 (work in progress), February 2011.

7.2. Informative References

[I-D.ietf-hybi-design-space] Moffitt, J., Loreto, S., Saint-Andre, P., and S. Salsano, "Design Space for Bidirectional Protocols", draft-ietf-hybi-design-space-00 (work in progress), June 2010.

Appendix A. Change Log (to be removed by RFC Editor before publication)

From version -01 to version -02:

- o In Req. 1, clarified that the transport protocol is TCP.
- o Explicit mention of masking as one of the mechanisms to use in order to mitigate cross-protocol attacks.
- o Moved Requirements Language to terminology section.
- o Got rid of MUST in intro.

From version -00 to version -01:

- o Modified definition of a Message to reflect recent consensus that it may comprise multiple frames.
- o Added definitions for WebSocket handshake, WebSocket communication channel and WebSocket sub-protocol.

- o Updated references to official IETF documents, moved "design-space" to informational.

- o Added a new requirement to support the ability to fragment a message into frames of a given length

- o Reworded Req 5 to reflect recent consensus on "binary+data" as well as common framing with explicit length indication

- o Reworded Req 7 to reflect recent consensus (declared in Maastricht) on "HTTP compliance"

- o Reworded Req 12 and 13 into a single Req on the client being able to send text or binary content. Elided mention of "discrete blocks" as the structure of messages is captured elsewhere.

- o Reworded Req 15 and 16 into a single Req on the server being able to send text or binary content. Elided mention of "discrete blocks" as the structure of messages is captured elsewhere.

- o Added a requirement in the proxies requirements section along the lines of the HTTP compliance requirement, per consensus declared in Maastricht.

- o Modified security requirement when used from outside a browser to avoid wording in terms of a security model equivalent to that of browser-based usage.

- o Various editorial changes.

Author’s Address

Gabriel Montenegro (editor)
Microsoft Corporation
One Microsoft Way
Redmond 98052
USA

Email: gabriel.montenegro@microsoft.com

HYBI
Internet-Draft
Intended status: Standards Track
Expires: January 18, 2013

M. Thomson
Microsoft
S. Loreto
Ericsson
G. Wilkins
Intalio

July 17, 2012

Hypertext Transfer Protocol (HTTP) Keep-Alive Header
draft-thomson-hybi-http-timeout-03

Abstract

A Keep-Alive header is defined for HTTP. This hop-by-hop header informs hosts about connection management policies. A parameters is defined for idle connection timeout.

Status of this Memo

Copyright Notice

Table of Contents

1. Introduction

This document describes the "Keep-Alive" header. The "Keep-Alive" header provides Hypertext Transfer Protocol (HTTP) [I-D.ietf-httpbis-p1-messaging] clients, servers and intermediaries with information about the connection use policies of their peers.

The "timeout" Keep-Alive parameter indicates the time that a connection will be allowed to remain idle before it is closed.

Some HTTP implementations already provide an implementation for this header. Not all of those implementations are interoperable due to significant differences in the header format. This draft defines a single format for the header and ascribes specific semantics to the header parameters.

1.1. Idle Connection Timeouts and Connection Reuse

Management of idle HTTP connections has an impact on long-lived communications between hosts. Hosts are able to close idle connections in order to reduce resource consumption.

Many clients choose not to send non-idempotent requests on idle connections. If the intermediary or server holding the other end of the connection chooses to close the connection while a non-idempotent request is in transit, the client has no way to tell if the request has succeeded. For this reason, many clients establish a new connection for every non-idempotent request. This is inefficient if the existing connection is a usable connection: establishing a new connection adds significantly to the latency of the request.

Connection resources can be more efficiently used when an idle connection timeout is known. A client that only periodically sends request can learn about the possibility of a connection timeout and can act to create a new connection for requests or send requests that keep the connection from timing out. Alternatively, a client that knows that more requests on a connection are unlikely within the discovered timeout interval can close the connection immediately after a poll, releasing resources.

1.2. Terminology

In this document, the key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" are to be interpreted as described in BCP 14, RFC 2119 [RFC2119] and indicate requirement levels for compliant implementations.

2. Keep-Alive Header

The "Keep-Alive" header is a hop-by-hop header that provides information about a persistent connection. Both client and server are able to provide information independently.

```
Keep-Alive           = "Keep-Alive" ":" 1#keep-alive-info
keep-alive-info      =   "timeout" "=" delta-seconds
                       / keep-alive-extension
keep-alive-extension = token [ "=" ( token / quoted-string ) ]
```

This header is sent by either host participating in a persistent connection. The values might be set based on policy implemented by servers, clients and intermediaries. Values might also be set based on knowledge that a host has about lower layer intermediaries in the path of the request, such as a TCP middlebox. Such middleboxes, in particular network address translators (NATs), frequently discard mappings for idle connections, causing the connection to fail after a certain duration of inactivity.

The value of Keep-Alive parameters can change on each request or response sent on a connection. Absence of the header or any parameter implies that any previously provided value still applies.

As a hop-by-hop header, this header only applies to a single transport-level connection. If a Keep-Alive header is added to a request or response, the Connection header MUST include the tag "Keep-Alive". This ensures that compliant intermediaries that do not recognize this header remove it before forwarding a request.

2.1. 'timeout' Parameter

A host sets the value of the "timeout" parameter to the time that the host will allow an idle connection to remain open before it is closed. A connection is idle if no data is sent or received by a host.

The value of the "timeout" parameter is a single integer in seconds.

A host MAY keep an idle connection open for longer than the time that it indicates, but it SHOULD attempt to retain a connection for at least as long as indicated.

Each peer, client or server, has a different view of the time that a connection becomes idle. Packet transmission at one peer necessarily occurs before receipt, meaning that the sending peer perceives the connection as being idle earlier to the receiving peer. Similarly, the buffering or retransmission of data by lower layers of the stack, which is unlikely to be visible to the HTTP implementation, compounds this effect. Clients are advised to make allowances for delays in determining whether to reuse an idle connection.

2.2. Other Header Parameters

The Keep-Alive header can be extended by adding any number of keep-alive-extension values to the header. Any extension that is not understood MUST be ignored.

The HTTP Keep-Alive Information Registry defines the namespace for Keep-Alive extensions. Section 7.2 describes this registry.

2.2.1. 'max' Parameter

The "max" parameter has been used to indicate the maximum number of requests that would be made on the connection. This parameter is deprecated. Any limit on requests can be enforced by sending "Connection: close" and closing the connection.

3. Existing Intermediaries

The exact impact of an intermediary on an HTTP request with a Keep-Alive header depends on the type of intermediary.

An intermediary that is compliant with HTTP/1.1, but does not implement Keep-Alive, ignores and discards this header before forwarding a request. Since it is unaware of the semantics of the header it could drop an idle connection at any time (see Section 7.1.4 of [I-D.ietf-httpbis-p1-messaging]).

A non-compliant "transparent" intermediary could pass this header on to the next hop. This results in errors of the sort that are described in Section A.1.2 of [I-D.ietf-httpbis-p1-messaging]. Clients that send this header to HTTP/1.0 servers or proxies SHOULD monitor for "hung" connections and avoid sending the header if a connection appears to hang.

A network address translation (NAT) device or other middlebox might cause a connection to become unavailable prior to the advertised timeout.

A client or intermediary can revise or remove the Keep-Alive header for subsequent requests to the same resource or origin server if it detects non-compliant intermediaries or middleboxes that have shorter timeout periods.

4. Upgraded HTTP Connections

A connection timeout can apply to a connection that is subsequently upgraded to another protocol [RFC2817], such as the websocket protocol [RFC6455].

The idle connection timeout applies to the upgraded connection, unless the upgraded protocol provides another method for indicating idle timeouts.

A server, client or intermediary might apply different policies to an upgraded protocol.

Upgraded protocols might establish an end-to-end connection. As a hop-by-hop header, the values in the "Keep-Alive" on each hop apply to every hop equally. For "timeout", this means that the lowest value from any hop applies to the connection.

Intermediaries that support this header SHOULD determine the impact of a header parameter on dependent hops and reflect that in the values they set. For "timeout", this means that the lowest value from the values seen and the local value is provided in outgoing messages.

5. Examples

The example in Figure 1 shows how a Keep-Alive header could be used. All connections are independently negotiated. In this example, the client indicates a timeout of 600 seconds (10 minutes), but the proxy is prepared to retain the connection for up to 3600 seconds (1 hour). On the link between proxy and server, the proxy requests a timeout of 1200 seconds and the server indicates a lower limit of 300 seconds.

```
 Client                         Proxy                          Server
   |                              |                              |
   +- Keep-Alive: timeout=600 -->|                              |
   |  Connection: Keep-Alive     |                              |
   |                             +- Keep-Alive: timeout=1200 -->|
   |                              |  Connection: Keep-Alive      |
   |                              |                              |
   |                              |<-- Keep-Alive: timeout=300 --+
   |                              |    Connection: Keep-Alive    |
   |<- Keep-Alive: timeout=3600 -+                              |
   |    Connection: Keep-Alive    |                              |
   |                              |                              |
```

Figure 1: Independent HTTP Hops

As this example shows, the timeout policies maintained by the proxy are different for each connection. Each connection hop is independent.

The example in Figure 2 shows the headers included in an upgrade from HTTP/1.1 to WebSocket [RFC6455]. With a websocket upgrade, the connections on each hop cannot have independent lifecycles on either side of an intermediary. After the upgrade, timeout policies cannot be independent for each hop. The proxy adjusts the timeout value to reflect the lower of the values set by client and the proxy policies so that the server is aware of the connection characteristics; similarly, the value from the server is provided to the client.

```
 Client                        Proxy                        Server
   |                             |                             |
   |  Upgrade: websocket         |                             |
   +- Keep-Alive: timeout=600 -->|                             |
   |  Connection: Keep-Alive,    |                             |
   |              Upgrade        |                             |
   |                             |  Upgrade: websocket         |
   |                             +- Keep-Alive: timeout=600 --->|
   |                             |  Connection: Keep-Alive,    |
   |                             |              Upgrade        |
   |                             |                             |
   |                             |    Upgrade: websocket       |
   |                             |<-- Keep-Alive: timeout=300 --+
   |                             |    Connection: Keep-Alive,  |
   |                             |                Upgrade      |
   |   Upgrade: websocket        |                             |
   |<- Keep-Alive: timeout=300 --+                             |
   |   Connection: Keep-Alive,   |                             |
   |               Upgrade       |                             |
   |                             |                             |
```

Figure 2: Interdependent Connections with Upgrade

6. Security Considerations

Establishing a persistent connection requires a commitment of resources at a host. The Keep-Alive header are used to express host policy that could alter the way that a host allocates connection resources. Since these policies can be enacted without this feedback, these indications have little effect on security other than exposing specifics of policy.

A host can close a non-idle connection sooner than the indicated time if necessary or as dictated by local policy (see Section 7.1.4 of

[I-D.ietf-httpbis-p1-messaging]).

7. IANA Considerations

[[Note to IANA/RFC Editor: Please replace instance of RFCXXXX with the number of the published RFC and remove this note.]]

7.1. Registration for Keep-Alive Header

This document registers the HTTP "Keep-Alive" header in the "Permanent Message Header Fields" registry established by [RFC3864]

Header field: Keep-Alive

Applicable protocol: HTTP

Status: standard

Author/change controller: Internet Engineering Task Force, IETF (iesg@ietf.org)

Specification document(s): RFCXXXX (this document)

7.2. Registry for Keep-Alive Information

This document establishes a registry for Keep-Alive Information.

Registrations are subject to Specification Required [RFC5226]. The designated expert is advised to review registrations and work with the submitter to ensure that:

- the registration name conforms to the HTTP "token" grammar
- a stable specification exists that is sufficient for interoperable implementation
- the registration does not duplicate an existing entry

The registry includes the following initial values:

timeout See Section 2.1 of this document.

max Deprecated. See Section 2.2.1 of this document.

8. Acknowledgments

Jamie Lokier provided valuable contributions of experience, insight and text suggestions to this document. Roy Fielding provided information on existing implementations of the poorly documented header. Also provided useful feedback: Patrick McManus, Dave Thaler, Konstantinos Pentikousis.

9. Change Log

Since -01:

- o Deprecated 'max'
- o Corrected badly misleading examples
- o Loosened the registry policy from IETF Review

Since -00:

- o Removed Request-Timeout in favor of the wait parameter of the Prefer header.
- o Connection-Timeout has now been replaced with the zombie spawn of Keep-Alive. This means that it picks up the 'max' parameter as baggage. Open question: should 'max' be deprecated?

Since draft-loreto-http-timeout:

- o Changed Timeout to Request-Timeout to avoid a conflict with an existing header definition.
- o Added note about the application of Connection-Timeout to upgraded protocols.

10. References

10.1. Normative References

[I-D.ietf-httpbis-p1-messaging]
Fielding, R., Lafon, Y., and J. Reschke, "HTTP/1.1, part 1: Message Routing and Syntax"", draft-ietf-httpbis-p1-messaging-20 (work in progress), July 2012.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate

Requirement Levels", BCP 14, RFC 2119, March 1997.

[RFC3864] Klyne, G., Nottingham, M., and J. Mogul, "Registration Procedures for Message Header Fields", BCP 90, RFC 3864, September 2004.

[RFC5226] Narten, T. and H. Alvestrand, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 5226, May 2008.

10.2. Informative References

[RFC2068] Fielding, R., Gettys, J., Mogul, J., Nielsen, H., and T. Berners-Lee, "Hypertext Transfer Protocol -- HTTP/1.1", RFC 2068, January 1997.

[RFC2817] Khare, R. and S. Lawrence, "Upgrading to TLS Within HTTP/1.1", RFC 2817, May 2000.

[RFC6455] Fette, I. and A. Melnikov, "The WebSocket Protocol", RFC 6455, December 2011.

Authors’ Addresses

Martin Thomson
Microsoft
3210 Porter Drive
Palo Alto, CA 94304
US

Phone: +1 650-353-1925
Email: martin.thomson@skype.net

Salvatore Loreto
Ericsson
Hirsalantie 11
Jorvas 02420
Finland

Email: salvatore.loreto@ericsson.com

Greg Wilkins
Intalio
644 Emerson Street, Suite 200
Palo Alto 94301
USA

Email: gregw@intalio.com