A Multiplexing Extension for WebSockets
draft-ietf-hybi-websocket-multiplexing-10

Abstract

The WebSocket Protocol [RFC6455] requires a new transport connection for every WebSocket connection. This presents a scalability problem when many clients connect to the same server, and is made worse by having multiple clients running in different tabs of the same user agent. This extension provides a way for separate logical WebSocket connections to share an underlying transport connection.

Please send feedback to the hybi@ietf.org mailing list.

Status of this Memo

This Internet-Draft is submitted to IETF in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at http://datatracker.ietf.org/drafts/current.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on May 16, 2013.

Copyright Notice

Copyright (c) 2012 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust’s Legal Provisions Relating to IETF Documents (http://trustee.ietf.org/license-info) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must
include Simplified BSD License text as described in Section 4.e of
the Trust Legal Provisions and are provided without warranty as
described in the Simplified BSD License.

Table of Contents

1. Overview .................................................. 3
   1.1. Physical Connection and Logical Channels .......... 3
2. Conformance Requirements ............................... 4
3. Extension Negotiation ................................. 5
4. Interaction with other Extensions / Framing Mechanisms . 6
   4.1. Ordering Extensions ............................... 6
      4.1.1. Efficiency .................................. 6
      4.1.2. Security ................................... 7
5. Multiplexed Connections ............................... 8
6. Flow Control ............................................. 9
   6.1. New Channel Slot ................................ 9
   6.2. Send Quota ........................................ 9
7. Framing .................................................. 11
8. Encapsulation ............................................ 13
9. Multiplex Control Blocks ............................. 15
   9.1. Number Encoding in Multiplex Control Blocks ...... 16
   9.2. AddChannelRequest .............................. 17
   9.3. AddChannelResponse ............................ 20
   9.4. FlowControl .................................... 22
   9.5. DropChannel ..................................... 23
      9.5.1. Drop Reason Codes .......................... 24
   9.6. NewChannelSlot .................................. 27
10. Examples .................................................. 29
11. Client Behavior ......................................... 31
12. Buffering ............................................... 32
13. Fairness ............................................... 33
14. Proxies ............................................... 34
15. Timeout ................................................ 35
16. Close the Logical Channel ........................... 36
17. Fail the Logical Channel ............................. 37
18. Fail the Physical Connection ........................ 38
19. Operations and Events on Multiplexed Connection .... 39
20. Security Considerations .............................. 40
21. IANA Considerations ................................. 41
22. References ............................................. 42
   22.1. Normative References ............................ 42
   22.2. Informative References .......................... 42
Authors’ Addresses ........................................ 43
1. Overview

This document describes a multiplexing extension for the WebSocket Protocol. With this extension, one TCP connection can provide multiple virtual WebSocket connections by encapsulating frames tagged with a channel ID. A client that supports this extension will advertise support for it in the client’s opening handshake using the "Sec-WebSocket-Extensions" header. If the server supports this extension and supports parameters compatible with the client’s request, it accepts the use of this extension by the "Sec-WebSocket-Extensions" header in the server’s opening handshake.

1.1. Physical Connection and Logical Channels

Under use of this extension, one transport connection is shared by multiple application-level instances. The WebSocket connection which lies directly on the TCP connection and negotiated this multiplexing extension is called "physical connection". Virtual WebSocket connections established for each application-level instance are called "multiplexed connections". Data channels virtually established by ID tagging are called "logical channels". This extension assigns a non-zero ID for each multiplexed connection. Each logical channel with a non-zero ID exchanges frames of the multiplexed connection with the same ID. The logical channel with ID of 0 exchanges data to control multiplexing.

Data for different logical channels are distinguished by the logical channel ID tag field placed at the head of the message that encapsulates the original frame of a multiplex connection.
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", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC2119 [RFC2119].

Requirements phrased in the imperative as part of algorithms (such as "strip any leading space characters" or "return false and abort these steps") are to be interpreted with the meaning of the key word ("must", "should", "may", etc) used in introducing the algorithm.

Conformance requirements phrased as algorithms or specific steps MAY be implemented in any manner, so long as the end result is equivalent. (In particular, the algorithms defined in this specification are intended to be easy to follow, and not intended to be performant.)
3. Extension Negotiation

The registered extension token for this extension is "mux".

To request use of the WebSocket Multiplexing Extension, a client MUST include an element with the "mux" extension token as its extension identifier in the "Sec-WebSocket-Extensions" header in its opening handshake. The element MAY contain an extension parameter named "quota". The value of the "quota" extension parameter specifies the server's send quota for the "Implicitly Opened Connection".

To accept use of the WebSocket Multiplexing Extension, a server MUST include an element with the "mux" extension token in the "Sec-WebSocket-Extensions" header in its opening handshake. The element MUST NOT contain any extension parameter.

A server MAY choose to reject use of the WebSocket Multiplexing Extension by not including the element for the extension in the "Sec-WebSocket-Extensions" header in its opening handshake. If any elements were listed after the element for the WebSocket Multiplexing Extension in the "Sec-WebSocket-Extensions" from the client, they MUST also be rejected.
4. Interaction with other Extensions / Framing Mechanisms

If any extension (e.g. compression) is placed before this extension in the "Sec-WebSocket-Extensions" header of the physical connection, that extension is applied to multiplexed connections unless otherwise noted in the extension’s spec.

If any extension is placed after this extension in the "Sec-WebSocket-Extensions" header of the physical connection, that extension is applied to frames after multiplexing on the sender side, and before demultiplexing on the receiver side unless otherwise noted in the extension’s spec.

A client MAY request such an extension for both the physical connection and multiplexed connections by placing extension entries before and after this multiplexing extension. In this case, the server SHOULD reject at least either of them if it’s useless to apply the same extension twice.

For example, if we have a compression extension called foo-compress, the client sends

```
Sec-WebSocket-Extensions: foobar, mux, foobar
```

in the client’s opening handshake of the physical connection to request use of the foobar extension for both physical and multiplexed connections. Then, the server would send back

```
Sec-WebSocket-Extensions: mux, foobar
```

to apply the foobar extension after multiplexing, or

```
Sec-WebSocket-Extensions: foobar, mux
```

to apply the foobar extension to multiplexed connections.

4.1. Ordering Extensions

4.1.1. Efficiency

Where to apply a compression extension makes difference to resource consumption and flexibility. Compression algorithms often use some memory to keep its context. Some of compression extensions may keep using the same context for all the messages on the same connection.

If such a compression extension is applied to the physical connection, intermediaries that want to demultiplex or multiplex the connection need to decompress (before demultiplexing) and recompress...
(before multiplexing again) all the frames.

If such an compression extension is applied to each multiplexed connection, we can control to which multiplexed connection we apply the compression, so we can avoid applying compression to multiplexed connections transferring incompressible data. Intermediaries that want to demultiplex can forward payload leaving it untouched. However, compressing each multiplexed connection is expensive in terms of memory consumption.

4.1.2. Security

If any history-based compression extension such as DEFLATE is applied to the physical connection that is tunneled over Transport Layer Security (TLS) [RFC2818], it may spoil TLS’s confidentiality [CRIME]. If the client may run malicious script such as a web browser, it MUST NOT request use of the multiplexing extension and such a compression extension in the order by which the compression extension is applied to the physical connection side.
5. Multiplexed Connections

The multiplexing extension maintains separate logical channels, each of which provides fully the logical equivalent of an independent WebSocket connection, including separate handshake headers. If the multiplexing extension is successfully negotiated, one multiplexed connection is automatically established, and the headers on the opening handshake of the physical connection are automatically taken to mean ones for the multiplexed connection. It’s called "Implicitly Opened Connection". It’s served by the logical channel with ID of 1 which is also implicitly opened on completion of the opening handshake. New logical channels are added by the client issuing the AddChannelRequest multiplex control block (note that only the client may initiate new WebSocket connections), including any handshake headers which do not have the same value as the client’s opening handshake of the physical connection. The server’s AddChannelResponse likewise includes any handshake headers which are different from the server’s opening handshake of the physical connection Channel 0 (control channel) is reserved for multiplex control blocks and does not contain Payload Data from any multiplexed connection. A client which attempts to add a channel to an existing connection that is not accepted by the server SHOULD attempt to open a new underlying connection and open a new WebSocket connection on it.

Once the multiplexing extension is negotiated on a connection, all frames of multiplexed connection MUST be prefixed with a channel ID number and encapsulated into binary messages. The channel ID is assigned by the client on issuing an AddChannelRequest.

A receiver MAY process frames for different non-control channels in parallel. A receiver MUST process frames for the control channel exclusively.

A receiver MUST _Fail the Physical Connection_ if any of these rules are violated by the sender.
6. Flow Control

6.1. New Channel Slot

A client has a pool of slots called "new channel slots". It’s initialized to be empty on establishment of the physical connection.

A NewChannelSlot multiplex control block sent by the server adds slots to the pool.

Each slot has a non-negative integer value called "initial send quota". Its function is explained in the later subsection.

When sending an AddChannelRequest, the client MUST pick the oldest new channel slot from the pool and remove it from the pool. If there are no slots in the pool, the client MUST NOT issue an AddChannelRequest. An endpoint MUST _Fail the Logical Channel_ with drop reason code of 2007 when it’s clear that the other peer violates this.

A server can regulate the rate of AddChannelRequests by not replenishing the pool.

6.2. Send Quota

For each logical channel with non-zero ID, server and client are respectively given a non-negative integer value called "send quota".

For the logical channel created for the "Implicitly Opened Connection", the client’s "send quota" is initialized to 0 on establishment of the physical connection. The server’s "send quota" for the channel is initialized on sending its opening handshake for the physical connection. The "quota" extension parameter attached to the extension token for this multiplexing extension in the client’s opening handshake for the physical connection specifies the initial value. If the "quota" extension parameter is not specified, the initial value is set to 0. The extension parameter has the initial value on its parameter value side as a non-negative integer in decimal.

For a logical channel added by issuing an AddChannelRequest, a client gets "send quota" equal to the "initial send quota" value on the "new channel slot" picked for the AddChannelRequest on sending it.

For a logical channel added by accepting an AddChannelRequest, a server gets "send quota" of 0 on sending the corresponding AddChannelResponse.
When an endpoint receives a FlowControl for a logical channel, its "send quota" for the channel gets replenished.

An endpoint MUST NOT send a frame on a logical channel with non-zero ID while the "send quota" of the endpoint for the channel is less than the cost of the frame. The cost of a frame is sum of the following two values:

- The length of the "Payload data" of the frame.
- Per-message extra cost. It’s 1 if the frame is the first fragment of a message. Otherwise, it’s 0.

An endpoint MUST _Fail the Logical Channel_ with drop reason code of 3005 when it’s clear that the other peer violates this.

When a frame is sent on a logical channel with non-zero ID, the length of the "Payload data" of the frame is subtracted from the "send quota" of the endpoint for the channel.

An endpoint SHOULD NOT delay replenishment of the other peer’s "send quota" for a logical channel when it has more room for accepting new data for the channel.
The multiplexing extension uses binary messages to transfer both data for controlling multiplexing and data of multiplexed connections. Binary messages have the logical channel ID tag field at the head of them. Logical channel ID of 0 is designated for control channel where multiplex control blocks are exchanged. Non-zero logical channel IDs are used for non-control channels transferring data for multiplexed connections.

The ID in the logical channel ID tag field is encoded as variable number of bytes (1, 2, 3 or 4 octets), as follows:

```
0 1 2 3 4 5 6 7
+-+-------------+
|0|Channel ID(7)|
 +-+-------------+
```

```
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5
+-+---------------------------+
|1|0|      Channel ID (14)      |
 +-+---------------------------+
```

```
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3
+-+-+-----------------------------------------+
|1|1|0|             Channel ID (21)             |
 +-+-+-----------------------------------------+
```

```
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
+-+-+-+---------------------------------------------------------+
|1|1|1|                     Channel ID (29)                     |
 +-+-+-+---------------------------------------------------------+
```

This encoding is also used by multiplex control blocks where they need to specify the ID of the objective channel.

A field using this encoding is considered to be invalid when more than the minimal number of bytes necessary to represent the integer is used.

Unless any other negotiated extension defines a meaning for them, endpoints MUST NOT send any data message other than "binary". When an endpoint received such a frame, it MUST _Fail the Physical Connection_ with drop reason code of 2001.
When an endpoint received a binary message with an incomplete or invalid logical channel ID tag field at the head of the message, _Fail the Physical Connection_ with drop reason code of 2002.

See Section 8 (non-control channel) and Section 9 (control channel) for more details on fields that follow the logical channel ID tag field.
8. Encapsulation

This extension encapsulates each frame of a multiplexed connection into a binary message with Payload Data obtained by concatenating the following data in the order they are listed:

1. The logical channel ID tag field representing the ID of the logical channel for the multiplexed connection.

2. FIN, RSV1, RSV2, RSV3 and opcode of the original frame.

3. Unmasked "Payload Data" of the original frame.

These messages are called encapsulating messages.

A receiver MUST restore the original frame from the data and deliver to the corresponding multiplexed connection based on the ID in the logical channel ID tag field in the order they are received.

This extension MAY change the fragmentation of the original message before encapsulation in order to insert multiplex control blocks or adjust the amount of data to flush along with flow control.

When received an encapsulated frame with non continuation data opcode though the last data message of that logical channel has not yet been terminated by a fragment with the FIN bit set, the endpoint MUST _Fail the Logical Channel_ with drop reason code of 3009.

When received an encapsulated frame with the continuation opcode
though there’s no preceding message that has not yet been terminated on that logical channel, the endpoint MUST _Fail the Logical Channel_ with drop reason code of 3009.

On logical channels, control messages MAY also be fragmented. Fragmented control messages MUST be delivered to the corresponding multiplexed connection after receiving all fragments and defragmenting them. For non-first fragments of a control message, the continuation opcode (%x0) MUST be used for the opcode field as well as data messages. Fragments for any other message MUST NOT be injected between fragments of a control message on the same logical channel. When a demultiplexer received an encapsulated frame with a control opcode and the FIN bit unset, it MUST process encapsulated frames on the same logical channel with the continuation opcode following the encapsulated frame as fragments of the control message until it encounters one with the FIN bit set. When a demultiplexer encountered any encapsulated frame whose opcode is not continuation injected between fragments of a control message on the same logical channel, the endpoint MUST _Fail the Logical Channel_ with drop reason code of 3009.

To allow for adjustment of fragmentation, this multiplexing extension MUST NOT be used after any extension that does any of the followings:

- Require frame boundary on its output to be preserved.
- Use the "Extension data" field or any of the reserved bits on the WebSocket header as per-frame attribute.

Intermediaries that doesn’t understand the WebSocket Multiplexing Extension MAY fragment the encapsulating messages.

When received a binary message with a non-zero logical channel ID of an inactive channel (e.g. no channel has been opened for the logical channel ID, or the channel has been closed (by a DropChannel or an AddChannelResponse with the failure bit set) and not yet reopened), the endpoint MUST ignore the message.

When received a binary message with a non-zero logical channel ID which contains no octets in its payload after octets for the logical channel ID tag field, the endpoint _Fail the Physical Connection_ with drop reason code of 2003.
9. Multiplex Control Blocks

A binary message with the logical channel ID of 0 contains zero or more multiplex control blocks in "Payload data".

```
0 1 2 3 4 5 6 7
+---------------+  
| Channel ID of 0 |  
+---------------+  
| 1st multiplex  |  
| : control block :  
|                 |  
+---------------+  
| 2nd multiplex  |  
| : control block :  
|                 |  
+---------------+  
| ...            |  
| :             |  
|               |  
+---------------+  
```

Putting multiple control blocks into one WebSocket message saves framing overhead.

Each multiplex control block has fields as follows:

```
0 1 2 3 4 5 6 7
+--------------+  
| Opc        |  
|            +  
| Opc specific :  
| : data        |  
|               |  
+--------------+  
```

Opc

A multiplex control opcode as defined in the following subsections. Opc of 5-7 are reserved for future use.

Opc specific data

Data interpreted according to that opcode.

Each of the following subsections describes one multiplex control opcode and how to interpret opc specific data for that opcode.
If any reserved opcode is set to opc, the endpoint MUST _Fail the Physical Connection_ with drop reason code of 2004.

If any truncated multiplex control block is found, the endpoint MUST _Fail the Physical Connection_ with drop reason code of 2005 unless _Fail the Physical Connection_ is already done for any other error.

RSVs in the field diagrams of multiplex control blocks in this section means reserved bits. If any multiplex control block with any of the reserved bits set is found, the endpoint MUST _Fail the Physical Connection_ with drop reason code of 2005 unless _Fail the Physical Connection_ is already done for any other error.

An endpoint MAY process one or more multiplex control blocks without any error that are in the same message together with one with an error. This means that an endpoint doesn’t have to inspect all the blocks in the received message before starting processing blocks. An endpoint MAY also skip processing multiplex control blocks without any error that are in the same message together with one with an error.

9.1. Number Encoding in Multiplex Control Blocks

In addition to the logical channel ID encoding defined in the Section 7, we reuse the number encoding defined for payload length in the Section 5.2 of [RFC6455] for multiplex control blocks with a little modification. We call this number encoding "1/3/9 number encoding". Integers up to 0x7D MUST be encoded into 1 octet field containing the integer as is. Integers from 0x7E to 0xFFFF MUST be encoded into an octet of 0x7E followed by two octets containing the integer in network byte order. Integers from 0x10000 to 0x7FFFFFFFFFFFFFF MUST be encoded into an octet of 0x7F followed by eight octets containing the integer in network byte order. A field using the 1/3/9 number encoding is considered to be invalid when any of the following conditions is violated.

- The most significant bit of the first octet MUST be 0.
- The minimal number of bytes necessary to represent the integer MUST be used.
- If the first byte is 0x7F, the most significant bit of the next octet MUST be 0.

When received a multiplex control block with an invalid field using the 1/3/9 number encoding, the endpoint MUST _Fail the Physical Connection_ with drop reason code of 2005 unless _Fail the Physical Connection_ is already done for any other error.
9.2. AddChannelRequest

AddChannelRequest is sent only by clients to create a new logical channel, as if a new WebSocket connection were received on a separate transport connection.

When a client received an AddChannelRequest, it MUST _Fail the Physical Connection_ with drop reason code of 2005 unless _Fail the Physical Connection_ is already done for any other error.

Multiplex control opcode of AddChannelRequest is 0.

AddChannelRequest has fields as follows:

```
 0 1 2 3 4 5 6 7
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|0|0|0| RSV |Enc|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|Objective        |
 |channel ID |
 |(1-4 octet) |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|Handshake size |
 |(1-9 octet) |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|Handshake |
 |       |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

Enc

Handshake encoding type:

0 - identity

The client’s handshake data in the handshake field are sent as-is without any special encoding or compression applied, and constitute the complete set of a Request-Line, headers and the CRLF after the last header that would have been sent on opening handshake.

1 - delta-encoded

The client’s handshake data in the handshake field are delta-encoded, where any header that is not given is assumed to have the same value as that given on the current delta base.
The delta base is initialized to the client’s opening handshake of the physical connection but after modifying some headers as follows:

+ Remove the token for this extension and following ones from "Sec-WebSocket-Extensions". After this step, "Sec-WebSocket-Extensions" contains only extensions applied to the multiplexed connection.

+ Remove Upgrade, Sec-WebSocket-Key and Sec-WebSocket-Version header. They are not meaningful under multiplexing.

Every time, an AddChannelRequest where Enc field is identity is received, the delta base is updated to its handshake after this modification. The Request-Line MUST be sent regardless if it’s the same as one in the delta base or not.

For multiple headers with the same header name, multiplexer and demultiplexer MUST preserve the order in which they appeared. If any header which had multiple header lines in the delta-base is inherited, a demultiplexer MUST recover them in the same order they appeared in the delta-base.

A header with an empty value means that the header is not inherited from the delta base. If there were multiple headers with the specified name, none of them is inherited. When to send valueless headers, identity encoding MUST be used.

2-3 - reserved

Reserved for future use.

An endpoint MUST _Fail the Physical Connection_ with drop reason code of 2010 if any of reserved type is specified.

Objective channel ID

The ID of the logical channel objective to this operation. Encoding is the same as one used for the logical channel ID tag field. An endpoint MUST _Fail the Physical Connection_ with drop reason code of 2005 if this field is invalid.

Handshake size

The size of the handshake field encoded by the 1/3/9 number encoding.
Handshake

The client’s opening handshake as defined in Section 4 of RFC 6455 [RFC6455] for the new multiplexed connection. Upgrade, Sec-WebSocket-Key and Sec-WebSocket-Version header are excluded. This field is encoded using the encoding specified by the Enc field. An endpoint MUST _Fail the Physical Connection_ with drop reason code of 2009 if any problem is found in parsing this field.

If the logical channel ID specified by an AddChannelRequest is in use (including 0 for the control channel), it MUST _Fail the Physical Connection_ with drop reason code of 2006.

To accept an AddChannelRequest, the endpoint MUST send an AddChannelResponse with the failure bit unset and the objective channel ID field set to the objective channel ID specified in the AddChannelRequest. In this case, the channel becomes active.

To respond to an AddChannelRequest with status meaning handshake failure, the endpoint MUST send an AddChannelResponse with the failure bit set and its objective channel ID field set to the objective channel ID specified in the AddChannelRequest. In this case, the channel stays inactive.

An endpoint MAY reject an AddChannelRequest also by doing _Fail the Logical Channel_ with drop reason code of 3000. In this case, the channel stays inactive.

A server MAY delay responding to an AddChannelRequest and proceed to process subsequent multiplex control blocks or frames for multiplexed connections.

Channel ID assignment is done by client side. A client MAY use any algorithm to choose logical channel IDs for new channels. Note that logical channel ID assignment might be changed by intermediaries, so it’s not guaranteed that the value of logical channel ID is the same on the other peer.

Different from non-multiplexed WebSocket connection, a client MAY send frames of multiplexed connections except for "Implicitly Opened Connection" before receiving AddChannelResponse as far as there’s sufficient send quota. In case the AddChannelRequest fails, those frames are discared by the other peer. This doesn’t mean that users of this protocol such as the WebSocket API are required to allow their users to send frames before receiving the server’s opening handshake.
9.3. AddChannelResponse

AddChannelResponse is sent only by servers in response to the AddChannelRequest.

When a server received an AddChannelResponse, it MUST _Fail the Physical Connection_ with drop reason code of 2005 unless _Fail the Physical Connection_ is already done for any other error.

AddChannelResponse has fields as follows:

```
  0 1 2 3 4 5 6 7
+-+-+-+-+-+-+-+-+-
|0|0|1|F|RSV|Enc|
+-+-+-+-+-+-+-+-+-
|Objective      |
 :channel ID     :
 |(1-4 octet)    |
+-+-+-+-+-+-+-+-+-
|Handshake size |
 :(1-9 octet)    :
 |               |
+-+-+-+-+-+-+-+-+-
|Handshake      |
 :               :
 |               |
+-+-+-+-+-+-+-+-+-

F

Failure bit.
```

If the failure bit is not set, then the server has accepted the AddChannelRequest. The handshake field MUST contain a response to the request made by the AddChannelRequest. In this case, the channel becomes active.

If the failure bit is set, then the server has rejected the AddChannelRequest and this SHOULD be treated exactly the same as if a separate connection was attempted and the connection was closed after receiving the server’s handshake. Enc MUST be set to identity in this case. The handshake field MUST contain a response to the request made by the AddChannelRequest. In this case, the channel stays inactive. The sender of the AddChannelResponse with the failure bit set doesn’t have to send a DropChannel following the AddChannelResponse.
Enc

Encoding type.

Types are the same as ones defined for the AddChannelRequest opcode (but replacing "AddChannelRequest" with "AddChannelResponse", and "Request-Line" with "Response-Line").

The delta base is initialized to the server’s opening handshake of the physical connection but after modifying some headers as follows:

* Remove the token for this extension and following ones from "Sec-WebSocket-Extensions". After this step, "Sec-WebSocket-Extensions" contains only extensions applied to the multiplexed connection.

* Remove Upgrade and Sec-WebSocket-Accept header. They are not meaningful under multiplexing.

An endpoint MUST _Fail the Physical Connection_ with drop reason code of 2012 if any of reserved type is specified.

Objective channel ID

Same as one in the AddChannelRequest. If an inactive channel is specified, the endpoint MUST ignore this AddChannelResponse.

An endpoint MUST _Fail the Physical Connection_ with drop reason code of 2005 if this field is invalid.

Handshake size

The size of the handshake field encoded by the 1/3/9 number encoding.

Handshake

The server’s opening handshake as defined in Section 4 of RFC 6455 [RFC6455] for this multiplexed connection. Upgrade and Sec-WebSocket-Accept header are excluded because they are not meaningful under multiplexing. This field is encoded using the encoding specified by the Enc field.

An endpoint MUST _Fail the Physical Connection_ with drop reason code of 2011 if any problem is found in parsing this field.
If the server’s opening handshake is validated, the client MUST take this as _The WebSocket Connection is Established_.

9.4. FlowControl

FlowControl is used to replenish the other peer’s send quota for the specified logical channel.

Multiplex control opcode of FlowControl is 2.

FlowControl has fields as follows.

```
0 1 2 3 4 5 6 7
 +++++++------+
|0|1|0| RSV   |
 +++++++------+
|Objective |
 :channel ID :
 |(8-32 bit) |
 +-----------+
|Replenished |
 :send quota :
 |(1-9 octet) |
 +-----------+
```

Objective channel ID

Same as one in the AddChannelRequest. If an inactive channel is specified, the endpoint MUST ignore this FlowControl. An endpoint MUST _Fail the Physical Connection_ with drop reason code of 2005 if this field is invalid.

Replenished quota

The number of bytes the receiver can have outstanding towards the sender of the FlowControl message. It’s encoded by the 1/3/9 number encoding.

An endpoint MUST _Fail the Logical Channel_ with drop reason code of 3006 if its send quota for the channel exceeds 0x7FFFFFFFFFFFFFFF when the replenished quota is added. The endpoint MAY delay this _Fail the Logical Channel_ operation to process following multiplex control blocks and encapsulating messages that don’t affect the logical channel. When received a FlowControl with an invalid value in the replenished quota field, the endpoint MUST _Fail the Physical Connection_ as specified above rather than taking it as overflow.
9.5.  DropChannel

DropChannel is used to close a logical channel.

Multiplex control opcode of DropChannel is 3.

DropChannel has fields as follows:

```
  0 1 2 3 4 5 6 7
 ++++++--------+
 |0|1|1|   RSV   |
 ++++++--------+
 |Objective   |
 :channel ID  :
 |(1-4 octet) |
 +------------+
 |Reason size |
 :reason size :
 |(1-9 octet) |
 +------------+
 |Reason      |
 :reason     :
 +-----------+
```

Objective channel ID

Same as one in the AddChannelRequest. An endpoint MUST _Fail the Physical Connection_ with drop reason code of 2005 if this field is invalid.

Reason size

The size of the reason field encoded by the 1/3/9 number encoding. A DropChannel block with 1-octet reason field MUST be considered as a truncated multiplex control block.

Reason

The reason of closure. Reason MAY be empty. If reason is not empty, the first two bytes MUST be a 2-byte unsigned integer (in network byte order) representing a drop reason code. Following the 2-byte integer, reason MAY contain UTF-8-encoded human readable drop reason phrase.

When an endpoint received a DropChannel for an active non-control channel, the endpoint MUST tear down the logical channel, and the application instance that used the logical channel MUST treat this as
closure of underlying transport.

When an endpoint received a DropChannel in response to an AddChannelRequest, the endpoint MUST abort creation of the logical channel, and the application instance that requested creation of the logical channel MUST treat this as closure of underlying transport without receiving reply for the creation request.

When an endpoint sent or received a DropChannel for an active non-control channel, the endpoint MUST mark the channel as inactive. If the endpoint is server and it has not already sent a DropChannel for the channel, it MUST send a DropChannel with drop reason code of 3008 so that the client can mark the ID of the channel available for a new AddChannelRequest.

Once received a DropChannel for a non-control channel, the ID of the logical channel becomes available again for a new AddChannelRequest.

9.5.1. Drop Reason Codes

Drop reason codes are 4 digit unsigned integers.

1000-1999 are for normal closure on a logical channel without any multiplexing level error. These codes are used for dropping non-control channels.

1000 Normal closure

DropChannel with this drop reason code is commonly sent when _Close the WebSocket Connection_ is made on the multiplexed connection.

2000-2999 are for errors that _Fail the Physical Connection_. These codes are used for dropping the control channel.

2000 Physical connection failed

Used if a more specific error is not available.

2001 Invalid encapsulating message

Received a data message with non binary opcode.

2002 Channel ID is truncated or invalid

Received an encapsulating message with a logical channel ID which is truncated or invalid.
2003 Encapsulated frame is truncated
   Received an encapsulating message that contains only the logical channel ID tag field with non-zero value.

2004 Unknown multiplex control opcode
   Encountered a multiplex control block with unknown multiplex opcode.

2005 Invalid multiplex control block
   Encountered an invalid multiplex control block. E.g. objective channel ID is truncated, reserved bit is raised.

2006 Channel already exists
   Received an AddChannelRequest for an active logical channel.

2007 New channel slot violation
   Received an AddChannelRequest though the other peer has no new channel slot.

2008 New channel slot overflow
   Received a NewChannelSlot that overflows the number of new channel slots.

2009 Bad request
   Received an AddChannelRequest with a malformed handshake.

2010 Unknown request encoding
   Received an AddChannelRequest with an unknown encoding type.

2011 Bad response
   Received an AddChannelResponse with a malformed handshake.

2012 Unknown response encoding
   Received an AddChannelResponse with an unknown encoding type.

3000-3999 are for errors that _Fail the Logical channel_. These codes are used for dropping non-control channels.
3000 Logical channel failed

Used if a more specific error is not available.

3005 Send quota violation

Received an encapsulating message exceeding send quota.

3006 Send quota overflow

Received a FlowControl that overflows send quota.

3007 Idle timeout

Terminating an idle logical channel.

3008 DropChannel acknowledged

Used for a DropChannel sent in response to received DropChannel. When a server received a DropChannel and it hasn’t sent any DropChannel for that logical channel, the server MUST send a DropChannel with this reason code so that the client can release the channel ID and reuse it for a new AddChannelRequest safely.

3009 Bad fragmentation

Received an encapsulating message with bad fragmentation that cannot be delivered to the corresponding multiplexed connection.

4000-4999 are for requesting the other peer to take some actions. These codes are used for dropping non-control channels.

4001 Use another physical connection

The server is requesting the client to open a new physical connection and use it than adding any more logical channel until receiving a NewChannelSlot. A client received this reason code SHOULD NOT issue an AddChannelRequest on this physical connection until receiving a NewChannelSlot.

4002 Busy

The server is requesting the client to stop issuing an AddChannelRequest until receiving a NewChannelSlot. A client received this reason code SHOULD NOT issue an AddChannelRequest on this physical connection until receiving a NewChannelSlot.
9.6. NewChannelSlot

NewChannelSlot is sent only by servers to add new slots to the client’s new channel pool.

When a server received an NewChannelSlot, it MUST _Fail the Physical Connection_ with drop reason code of 2005 unless _Fail the Physical Connection_ is already done for any other error.

Multiplex control opcode of NewChannelSlot is 4.

NewChannelSlot has fields as follows:

```
  0 1 2 3 4 5 6 7
+-+-+-+-+-------+-+
|1|0|0|  RSV  |F|
+-+-+-+-+-------+-+
|Number of slots|
:(1-9 octet)    |
|               |
+---------------+
|Initial send   |
:quota          |
|(1-9 octet)    |
+---------------+
```

F

Fallback bit.

If the fallback bit is false, normal slot is added.

If the fallback bit is true, fallback suggestion slot is added. Number of slots field and initial quota field MUST be 0 for fallback suggestion slot. When a client encounters a fallback suggestion slot, it MUST open a new physical connection and use it than adding any more logical channel on this physical connection until any normal slot is available.

When received a NewChannelSlot block with the fallback bit set and any of the number of slots field or the initial quota field is not zero, the endpoint MUST _Fail the Physical Connection_ with drop reason code of 2005 unless _Fail the Physical Connection_ is already done for any other error.
Number of slots

The number of slots to add. It’s encoded by the 1/3/9 number encoding. This value MAY be 0 when it makes sense.

Initial quota

The initial quota each of slots added by this NewChannelSlot gets. It’s encoded by the 1/3/9 number encoding.

When a client received a NewChannelSlot, the client MUST add new slots of the specified number. Each of new slots gets the specified initial send quota.
10. Examples

_This section is non-normative._

The examples below assume the handshake has already completed and the multiplexing extension was negotiated. Quotes are for clarity.

Frames of encapsulating messages from client to server MUST be masked. The examples below are not masked for simplicity.

0x82 0x0d 0x01 0x81 "Hello world"

This is a non-fragmented text message of "Hello world" on logical channel 1 encapsulated into a non-fragmented encapsulating message.

0x82 0x07 0x01 0x01 "Hello" 0x82 0x08 0x01 0x80 " world"

This is a text message of "Hello world" fragmented into two frames of "Hello" and " world" on logical channel 1 encapsulated into two non-fragmented encapsulating messages. A multiplexer may change fragmentation of a message before encapsulation like this so that frames of other logical channels (including the control channel) can be injected in the middle of the message.

0x82 0x07 0x01 0x01 "Hello" 0x82 0x05 0x02 0x81 "bye" 0x82 0x08 0x01 0x80 " world"

This example shows how data for two logical channels are interleaved. There’re three non-fragmented encapsulating messages. As explained in the previous example, the text message of "Hello world" is split into two frames before encapsulation. The first and third frame in this example contain each of the two fragments of the text message of "Hello world" on logical channel 1. The second frame contains a non-fragmented text message of "bye" on logical channel 2.

0x82 0x04 0x01 0x01 "Te" 0x82 0x04 0x01 0x09 "Pi" 0x82 0x04 0x01 0x80 "ng" 0x82 0x04 0x01 0x80 "xt"

A ping message "Ping" is injected in the middle of a text message "Text" on the original connection. The multiplexer fragmented the ping message due to some reason into two fragments.

0x02 0x07 0x01 0x81 "Hello" 0x80 0x06 " world"

Encapsulating messages output from the multiplexer can be fragmented by intermediaries without knowledge of the Multiplexing
Extension. This is an example of a fragmented encapsulating message. It’s equivalent to the first example as a message.

0x82 0x16 0x00 0x02 0x12 "GET / HTTP/1.1" 0x0d 0x0a 0x0d 0x0a

This is a message on the control channel carrying one AddChannelRequest. The first two octets are the WebSocket headers. The 3rd octet is logical channel ID field of 0. The 4th octet has opcode, RSV and Enc field. Objective channel ID is 2. The handshake is delta-encoded.
11. Client Behavior

When a client is asked to _Establish a WebSocket Connection_ by some WebSocket application instance, it MAY choose to share an existing WebSocket connection if all of the following are true:

- the multiplexing extension was successfully negotiated on that connection
- the scheme portions of the URIs match exactly
- the host portions of the URIs either match exactly or resolve to the same IP address (TBD: consider DNS rebind attacks)
- the port portions of the URIs (either explicit or implied by the scheme) match exactly
- the connection has an available logical channel ID

If the client chooses to share the existing WebSocket connection with multiplexing, it sends an AddChannelRequest as described above. If the AddChannelRequest is accepted, WebSocket frames may be sent over that logical channel as normal. If the server rejects the AddChannelRequest, the client SHOULD attempt to open a new physical WebSocket connection (for example, in a shared hosting environment a server may not be prepared to multiplex connections from different customers despite having a single IP address for them).
12. Buffering

There will be lots of small frames sent in this protocol (particularly replenishing send quotas), so a sender SHOULD attempt to aggregate multiplex control blocks into larger WebSocket frames. For data frames, a sender also SHOULD attempt to aggregate fragments into one packet of the underlying transport. However, care must be taken to avoid introducing excessive latency - the exact heuristics for delaying in order to aggregate blocks is TBD.
13. Fairness

A multiplexing implementation MUST ensure reasonable fairness among the logical channels. This is accomplished in several ways:

Receiver side

- The receiver MAY limit the send quota of a logical channel by not replenishing it to make sure that any logical channel doesn’t dominate the connection.

- Send quota for one logical channel SHOULD be determined considering the processing capacity (buffer size, processing power, throughput, etc.) of that logical channel. For example, when a logical channel with excess load cannot drain data from the connection smoothly, the other logical channels get stuck even when they have room of processing capacity. Unless there’s special need to give such a big quota for the channel, such condition just makes overall performance low.

Sender side

- The sender MUST use a fair mechanism for selecting which logical channel’s data to send in the next WebSocket message. Simple implementations may choose a round-robin scheduler, while more advanced implementations may adjust priority based on the amount or frequency of data sent by each logical channel.

- The sender MUST fragment a large message into smaller frames to prevent a large message in a logical channel occupying the physical connection and thus delaying messages in other logical channels.
14. Proxies

Proxies which do not multiplex/demultiplex are not affected by the presence of this extension -- they simply process WebSocket frames as usual. Proxies which filter or monitor WebSocket traffic will need to understand the multiplexing extension in order to extract the data from logical connections or to terminate individual logical connections when policy is violated. Proxies which actively multiplex connections or demultiplex them (for example, a mobile network might have a proxy which aggregates WebSocket connections at a single cell to conserve bandwidth to the main gateway) will require additional configuration (perhaps including the client) that is outside the scope of this document.
15. Timeout

When all the logical channels are closed, each endpoint MAY _Start the WebSocket Closing Handshake_ on the physical connection. Such _Start the WebSocket Closing Handshake_ operation SHOULD be delayed assuming the physical connection may be reused after some idle period.
16. Close the Logical Channel

To _Close the Logical Channel_, an endpoint MUST send a DropChannel multiplex control block with drop reason code of 1000.
17. Fail the Logical Channel

To _Fail the Logical Channel_, an endpoint MUST send a DropChannel multiplex control block with drop reason code in the range of 3000-3999, tear down the logical channel, and the application instance that used the logical channel MUST treat this as closure of underlying transport.
10. Fail the Physical Connection

To _Fail the Physical Connection_, an endpoint MUST send a DropChannel multiplex control block with objective channel ID of 0 and drop reason code in the range of 2000-2999, and then _Fail the WebSocket Connection_ on the physical connection with status code of 1011.
19. Operations and Events on Multiplexed Connection

When an endpoint is asked to perform any operation defined in the WebSocket Protocol except for _Close the WebSocket Connection_ by some application instance, the endpoint MUST perform the operation on the corresponding logical channel.

Any event on a logical channel except for _The WebSocket Connection is Closed_, MUST be taken as one for the corresponding application instance.

When an endpoint is asked to do _Close the WebSocket Connection_ by some application instance, it MUST perform _Close the Logical Channel_ on the corresponding logical channel.

When a DropChannel is received, or the physical connection is closed, it MUST be taken as _The WebSocket Connection is Closed_ event for the corresponding application instance(s).

What to set to _Extension In Use_ for each multiplexed connection is TBD.
20. Security Considerations

A client MUST be prepared to receive a NewChannelSlot with huge value on the number of slots field.

As noted in the Section 4.1.2, be careful in using combination of any compression extensions and this extension.
21. IANA Considerations

This specification is registering a value of the Sec-WebSocket-Extension header field in accordance with Section 11.4 of the WebSocket protocol [RFC6455] as follows:

Extension Identifier

mux

Extension Common Name

Multiplexing Extension for WebSockets

Extension Definition

This document

Known Incompatible Extensions

None
22. References

22.1. Normative References


22.2. Informative References

Authors’ Addresses

John A. Tamplin
Google, Inc.

Email: jat@jaet.org

Takeshi Yoshino
Google, Inc.

Email: tyoshino@google.com