HyBi Working Group J. Tamplin
Internet-Draft T. Yoshino
Intended status: Standards Track Google, Inc.
Expires: August 31, 2012 February 28, 2012

A Multiplexing Extension for WebSockets
draft-tamplin-hybi-google-mux-03

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 August 31, 2012.

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 and Logical Channel 3
2. Conformance Requirements 4
3. Interaction with other Extensions / Framing Mechanisms 5
3.1. Choosing the point to apply an extension 6
4. Logical Channels 7
5. Flow Control 8
6. Framing 9
7. Multiplex Control Frames 11
7.1. Multiplex Control Opcodes 12
8. Examples 16
9. Client Behavior 17
10. Buffering 18
11. Fairness 19
12. Proxies 20
13. Nesting 21
14. Timeout 22
15. Close the Logical Channel 23
16. Fail the Logical Channel 24
17. Fail the Physical Channel 25
18. Handling Operations On Logical Channel 26
19. Security Considerations 27
20. IANA Considerations 28
21. Normative References 29
Authors' Addresses 30

1. Overview

This document describes a multiplexing extension for the WebSocket Protocol. 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 and Logical Channel

Under this extension, one transport connection is shared by multiple application-level instances. The WebSocket connection which lies directly on the transport connection and negotiated this multiplexing extension is called "physical channel". Virtually established WebSocket connections for each WebSocket appplication instances are called "logical channels".

Data for different logical channels are distinguished by the channel ID allocated in the "Extension data" portion of each frame.

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. Interaction with other Extensions / Framing Mechanisms

If WebSocket payload data is masked by a per-frame key, such masking is applied to frames for each logical channel separately.

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

If such an extension define fields in the "Extension data", they come after this multiplexing extension's field.

If any extension is placed after this extension in the "Sec-WebSocket-Extensions" header of the physical channel, 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.

If such an extension define fields in the "Extension data", they come before this multiplexing extension's field.

A client MAY request such an extension for both the physical channel and the logical channels 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: foo-compress, mux, foo-compress
```

in the client's opening handshake of the physical channel to request use of the compression for both physical and logical channels. Then, the server would send back

```
Sec-WebSocket-Extensions: mux, foo-compress
```

to apply compression after multiplexing, or

```
Sec-WebSocket-Extensions: foo-compress, mux
```

to apply compression to logical channels.

3.1. Choosing the point to apply an extension

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 frames on the same connection.

If such an extension is applied to the physical channel, 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 extension is applied to each logical channel, we can control to which channel we apply the compression, so we can avoid applying compression to channels transferring incompressible data. Intermediaries that want to demultiplex can forward Application data field leaving it untouched. However, compressing each logical channel is expensive in terms of memory consumption.

4. Logical Channels

The multiplexing extension maintains separate logical channels, each of which is fully the logical equivalent of an independent WebSocket connection, including separate handshake headers. If the multiplexing extension is successfully negotiated, the headers on the opening handshake of the physical channel are automatically taken to mean one for the logical channel 1, which is implicitly opened by completing the handshake. New channels are added by the client issuing the AddChannel request (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 channel. The server's AddChannel response likewise includes any handshake headers which are different from the server's opening handshake of the physical channel (the details of this are TBD, but a simple suggestion for a delta encoding is given below). Channel 0 (control channel) is reserved for multiplex control frames and does not contain payload data from any logical channel. In interpreting "Sec-WebSocket-Extensions" header for a logical channel, the entry for this multiplexing extension is ignored but is used to adjust parameters for the logical channel. 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 WebSocket connection.

If any inconsistency is found between the "Sec-WebSocket-Extensions" header for the physical channel and one for a logical channel (after decoding header compression), the server MUST reject the AddChannel request.

Once the multiplexing extension is negotiated on a connection, all frames must be prefixed with a channel ID number in the "Extension data". Control frames with a channel ID 0 refer to the physical channel, other control frames MUST be delivered on the logical channel in order with data frames for that logical channel. Control frames SHOULD be sent only on channel 0 where possible, though control frames for other extensions in particular may need to apply to individual logical channels.

A receiver MUST _Fail the Physical Channel_ if any of these rules are violated by the sender.

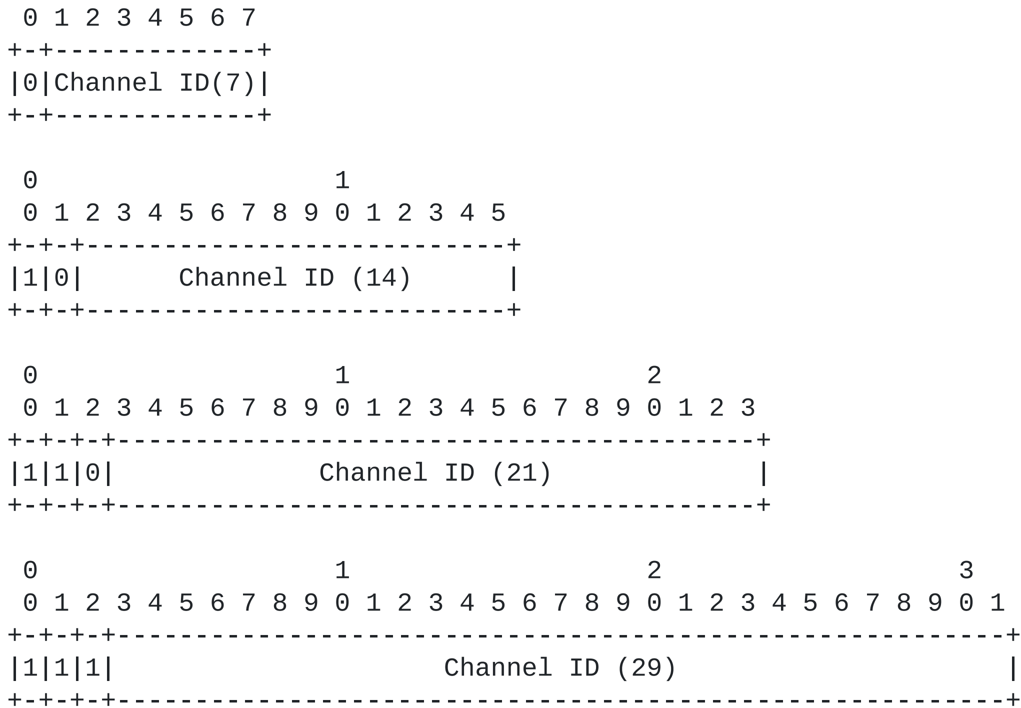
5. Flow Control

Each logical channel, including the implicitly created channel 1, is initially given a quota of bytes that may be transmitted in each direction without acknowledgement. It is illegal to send more bytes than the remaining send quota, and the receiver MUST _Fail the Logical Channel_ for any sender that does so. This send quota is replenished via control frames as the receiver processes the data.

The initial send quota is specified with the "quota" extension parameter, and defaults to 64k (TBD) if it is not specified. The client and server each may specify a "quota" parameter and these are unrelated -- each specifies how many bytes the other side may send without acknowledgement. The quota values in the opening handshakes of the physical channel apply to the implicitly opened channel 1.

6. Framing

If the extension is successfully negotiated during the opening handshake, all frames have a channel ID in the "Extension data". The channel ID is encoded as a variable number of bytes, as follows:



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

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

 0                   1                   2
 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
 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)                     |
+-+-+-+---------------------------------------------------------+
```

The base spec requires that a sequence of frames on the wire be a sequence of valid fragments (or one of valid unfragmented frames). The multiplexing extension relaxes this requirement to be for just frames of one logical channel, and that frames of other logical channels may be interleaved arbitrarily.

All frames with a non-zero channel ID must be delivered to the specified logical channel in the order they are received, though fragmentation may be changed if appropriate. Control frames with a non-zero channel ID may also trigger additional processing by the multiplexing extension.

Control frames with a channel ID of 0 refer to the physical connection, and may also trigger additional processing - for example, a close frame on the physical channel will close all logical channels as well (details TBD).

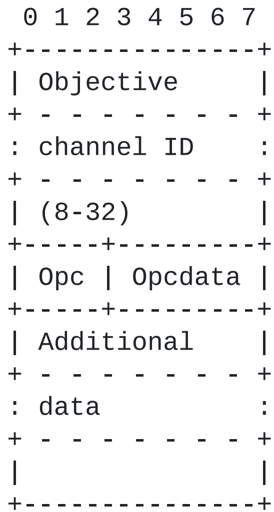
If a frame doesn't contain valid channel ID, _Fail the Physical Channel_. The cases where it's considered that the channel ID is

invalid are:

- The "Payload data" portion doesn't contain a complete channel ID.
- No channel has been opened for the channel ID.
- The channel has been closed and not reopened.

7. Multiplex Control Frames

Binary frames with a channel ID of 0 are multiplex control frames. Unless another negotiated extension defines a meaning for them, any data frames on channel 0 with an opcode other than "binary frame" MUST _Fail the Physical Channel_ "Payload data" of a multiplex control frames consists of a zero or more multiplex control blocks, each defined as follows:



```
  0 1 2 3 4 5 6 7
 +---------------+
 | Objective     |
 + - - - - - - - +
 : channel ID    :
 + - - - - - - - +
 | (8-32)        |
 +-----+---------+
 | Opc | Opcdata |
 +-----+---------+
 | Additional    |
 + - - - - - - - +
 : data          :
 + - - - - - - - +
 |               |
 +---------------+
```

Objective channel ID

The channel ID of the logical channel objective to this operation. Encoding is the same as that in the extension data (designated as control channel)

opc

A multiplex control opcode as defined in Section 7.1.

opcdata

Data interpreted according to that opcode

Additional data

Zero or more bytes defined by that opcode

If any incomplete multiplex control block is found, _Fail the Physical Channel_.

7.1. Multiplex Control Opcodes

0 - AddChannel request (only from client)

Create a new logical channel, exactly as if a new connection were received on a separate transport connection, except for the encoding of the headers. opcdata is interpreted as follows:

```
 3 4 5 6 7
+-+---+---+
|R|Enc|Len|
+-+---+---+
```

R is reserved for future use.

Len is the number of bytes used to represent the length of following handshake data minus 1.

Enc is an encoding scheme type:

0 - uncompressed

The handshake data that follow are uncompressed, and constitute the complete set of a Request-Line and headers that would have been sent on a WebSocket opening handshake

1 - delta-encoded

The handshake data that follow are delta-encoded, where any header that is not given is assumed to have the same value as that given on the client's opening handshake of the physical connection. The only exceptions are the Request-Line and the "Sec-WebSocket-Extensions" header. The Request-Line MUST be sent even if it's the same as one in the client's opening handshake for the physical channel. If the "Sec-WebSocket-Extensions" header is not given, its value is assumed to be the extension entry for this multiplexing extension and ones following that in the client's opening handshake of the physical channel. A header with an empty value means that header is not inherited from the initial connection. (TBD: this means that valueless headers cannot be encoded with this scheme).

2-3 - reserved

Reserved for future use

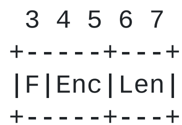
The following n bytes, where n is the value of len inside opcdata plus 1, are an 8-32 bit length of the client's opening handshake for the new logical channel that follows, in network byte order. It's encoded as defined by the enc value in opcdata.

The initial quota for the new logical channel is 0, so the client may not send any data for this connection until the AddChannel response is received.

The server always responds with an AddChannel response, described below.

1 - AddChannel response (only from server)

opcdata is defined as follows:



```
 3 4 5 6 7
+-----+---+
|F|Enc|Len|
+-----+---+
```

F is true if this response indicates a rejection of AddChannel request.

Len is the number of bytes used to represent the length of following handshake data minus 1.

Enc is an encoding scheme type defined as in the AddChannel request (but replacing Request-Line with Response-Line).

If F is set, then the server has rejected the AddChannel request and this SHOULD be treated exactly the same as if a separate connection was attempted and the opening handshake failed. Enc is ignored in this case, and the following n bytes, where n is the value of len inside opcdata plus 1, are an 8-32 bit length of the server's opening handshake for this logical channel that follows, in network byte order. It SHOULD be treated as the response to an HTTP Upgrade request for the request made by the AddChannel request, For example:

```
HTTP/1.1 404 Not found

404 message body...
```

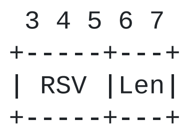
If F is not set, then the server has accepted the AddChannel request.

The following n bytes, where n is the value of len inside opcdata

plus 1, are an 8-32 bit length of the server's opening handshake for this logical channel that follows, in network byte order. It's encoded according to enc as defined in the AddChannel request, and the complete set of a Response-Line and headers after decoding is treated exactly as if it was received in response to a client's opening handshake on a separate connection. If the server's opening handshake is validated, the client MUST take this as _The WebSocket Connection is Established_.

2 - FlowControl

opcdata is defined as follows:



```
 3 4 5 6 7
+-----+---+
| RSV |Len|
+-----+---+
```

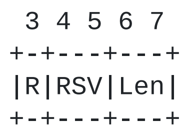
RSV is reserved for future use.

Len is the number of bytes used to represent the number of bytes to be added to the quota minus 1.

The following n bytes, treated as an unsigned integer in network byte order, is added to the quota of the number of bytes the receiver can have outstanding towards the sender of the FlowControl message. (TBD: is it worth having some non-linear encoding to reduce the average bits required to represent these values?)

3 - DropChannel

DropChannel is used to close a logical channel for both error cases and normal cases.



```
 3 4 5 6 7
+-+---+---+
|R|RSV|Len|
+-+---+---+
```

If R is set, it means that this DropChannel control block was sent due to _Fail the Logical Channel_. If R is unset, it means that this DropChannel control block was sent due to _Close the Logical Channel_.

RSV is reserved for future use.

Len is the number of bytes used to represent the length of following reason data minus 1.

The following n bytes, where n is the value of len inside opcdata plus 1, are an 8-32 bit length of the DropChannel reason string in network byte order.

When an endpoint received DropChannel, the endpoint MUST remove the logical channel and the application instance that used the logical channel MUST treat this as closure of underlying transport.

4-7 - reserved

Reserved for future use (TBD: do we need some support for quiescence?)

8. Examples

This section is non-normative.

The examples below assume the handshake has already completed and the x-google-mux extension was negotiated.

```
01 06 01 "Hello" 81 04 02 "bye" 80 07 01 " world"
```

This is a fragmented text message of "Hello world" on channel 1 interleaved with a text message of "bye" on channel 2. Note that the sequence of opcodes/FIN bits cannot be understood without considering the channel ID of each frame.

9. Client Behavior

When a client is asked to _Establish a WebSocket Connection_ by some WebSocket application instance, it MAY choose to reuse 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 availablle logical channel ID

If the client chooses to reuse an existing multiplexed connection, it sends an AddChannel request as described above. If the AddChannel request is accepted, WebSocket frames may be sent over that channel as normal. If the server rejects the AddChannel, 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).

10. 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.

11. 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 other peer's send quota of a logical channel by not replenishing the send quota to make sure that any logical channel cannot dominate its buffer space on the sender.

- 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 frame. 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 message into smaller frames when it's too big so that that logical channel will occupy the connection and the other logical channels get stuck for long time.

- Logical channel frames that are sent SHOULD be limited in size (such as by refragmenting) when there is contention for the physical channel to minimize head-of-line blocking

12. 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.

13. Nesting

TBD: Should we allow nesting of multiplexed channels, or should we require that an intermediary multiplexing channels flatten it? The advantage of nesting is it is conceptually cleaner and less work for an intermediary, while the disadvantage is that flow control messages will get amplified by nesting and the ultimate server's job is a bit more complicated to keep a tree of channel mappings.

14. 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 channel may be reused after some idle period.

15. Close the Logical Channel

To _Close the Logical Channel_, an endpoint MUST send a DropChannel multiplex control block with R bit unset. The endpoint MAY provide the reason of closure in the DropChannel block.

16. Fail the Logical Channel

To _Fail the Logical Channel_, an endpoint MUST send a DropChannel multiplex control block with R bit set. The endpoint MAY provide the reason of failure in the DropChannel block.

17. Fail the Physical Channel

To _Fail the Physical Channel_, an endpoint MUST send a DropChannel multiplex control block with objective channel ID of 0, and then _Fail the WebSocket Connection_ on the physical channel with status code of 1002 (TBD).

18. Handling Operations On Logical Channel

When an endpoint is asked to perform any operation defined in the WebSocket Protocol except for _Close the WebSocket Connection_ by some application instance, it MUST perform it 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 and the logical channel hasn't yet received DropChannel before that, it MUST be taken as _The WebSocket Connection is Closed_ event for the corresponding application instance.

19. Security Considerations

To protect a server from denial-of-service attack, implementation SHOULD have a way to limit the number of concurrent logical channels.

TBD

20. 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

Mulplexing Extension for WebSockets

Extension Definition

This document [draft-tamplin-hybi-google-mux] defines the mux extension.

Known Incompatible Extensions

None

21. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

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

Authors' Addresses

John A. Tamplin
Google, Inc.

Email: jat@google.com

Takeshi Yoshino
Google, Inc.

Email: tyoshino@google.com