HyBi Working Group
Internet-Draft
Intended status: Standards Track
Expires: July 30, 2012

J. Tamplin
T. Yoshino
Google, Inc.

January 27, 2012

A Multiplexing Extension for WebSockets
draft-tamplin-hybi-google-mux-02

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 July 30, 2012.

Copyright Notice

Table of Contents

1. Overview

This document describes a MUX extension to the WebSocket protocol. A client that supports this extension will advertise support for it in the initial 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 in the handshake response.

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

Any compression extensions negotiated apply only to the channel they are negotiated on -- therefore, any compression extension in the initial handshake applies only to logical channel 1. If WebSocket payload data is masked by a per-frame key, such masking is applied to frames for each logical channel separately.

If other negotiated extensions define extension data, the other extension defines whether it applies to just one logical channel (it is expected that most extensions will do so) or the physical channel. If the other extension applies to one logical channel, it always comes after the MUX extension data; otherwise the order depends on the order the extensions were listed during the handshake response.

4. Channels

The MUX extension maintains separate logical channels, each of which is fully the logical equivalent of an independent WebSocket connection, including separate handshake headers. If the MUX extension is successfully negotiated, the headers on the initial handshake are automatically taken to mean channel 1, which is implicitly opened by completing the handshake. New channels are added by the client issuing the AddChannel command (note that only the client may initiate new WebSocket connections), including any request headers which do not have the same value as the initial handshake. The server's AddChannel response likewise includes any response headers which are different from the initial handshake (the details of this are TBD, but a simple suggestion for a delta encoding is given below). Channel 0 (control channel) is reserved for mux control messages and does not contain payload data from any logical channel. A client which attempts to add a channel to an existing connection that is not accepted by the server SHOULD attempt a new WebSocket connection.

Once the MUX extension is negotiated on a connection, all frames must be prefixed with a channel number in the extension data field. Control frames with a channel id 0 refer to the physical connection, 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 connection and all open logical channels if any of these rules are violated by the sender.

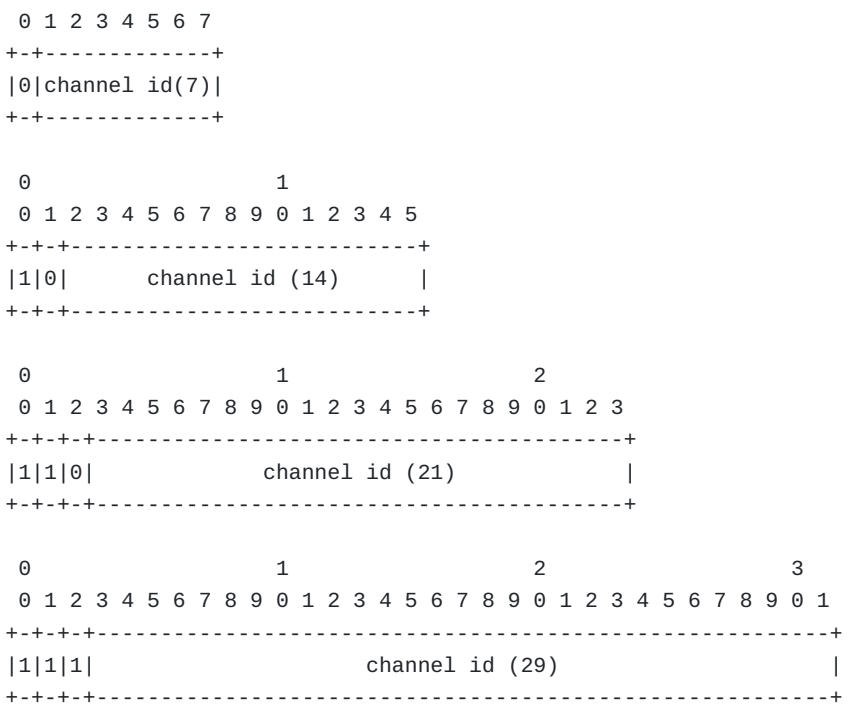
5. Flow Control

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 quota, and the receiver MUST fail the logical channel for any sender that does so. This quota is replenished via control messages as the receiver processes the data.

The initial 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 initial handshake apply to the implicitly opened channel 1.

6. Framing

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



The base spec requires that a sequence of frames on the wire be a frame with an opcode other than 0, zero or more frames with opcode 0 and the FIN bit clear, and terminated by a frame with the FIN bit set (which may be the initial frame in the case of an unfragmented message). The MUX 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 MUX extension.

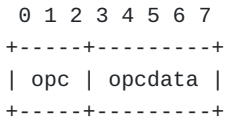
Control frames with a channel id of 0 refer to the physical connection, and may also trigger additional processing - for example, a close frame on the physical channel will close all logical channels as well (details TBD).

7. Multiplex Control Frames

Data frames with a channel id of 0 are MUX control frames. Unless another negotiated extension defines a meaning for them, any frames on channel 0 with an opcode other than "binary" MUST trigger a failure of the physical connection. Binary frames on channel 0 are MUX control frames, and the payload consists of a zero or more MUX control blocks, each defined as follows:

- a channel number encoded the same as that in the extension data (designated as control channel)

- an opcode and data interpreted according to that opcode as follows:

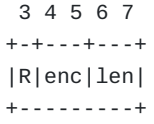


- zero or more bytes defined by the opcode

The opcodes defined are:

0 - AddChannel request (only from client)

Create a new logical channel as control 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:



R is reserved for future use, len is the number of bytes used to represent the length of following headers minus 1, and enc is an encoding type:

0 - uncompressed The header bytes that follow are uncompressed, and constitute the complete set of headers that would have been sent on a WebSocket connection request

1 - delta-encoded The header bytes that follow are delta-encoded, where any header that is not given is assumed to have the same value as that given on the initial request of the physical connection. 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

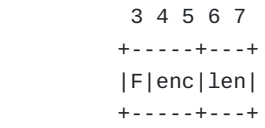
The following n bytes, where n is the value of len inside opcdata plus 1, are an 8-32 bit length of the request header data that follows, in network byte order. The request header data consists of a series of lines, separated by a CR-LF pair and terminated by an extra CR-LF pair. The first line is the full URI for the new connection, and the remaining lines are the request headers, encoded in UTF8 and as defined by the enc value in opcdata.

The initial quota for the new connection 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 message, described below.

1 - AddChannel response (only from server)

opcdata is defined as follows:



RSV is reserved for future use, F is true if this response indicates a failure to add the requested channel, len is the of bytes used to represent the length of following headers minus 1, and enc is an encoding scheme defined as in the AddChannel request.

If F is set, then the server has rejected the request to add a new channel and this should be treated exactly the same as if a separate connection was attempted and the 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 request header data that follows, in network byte order. The request header data consists of a series of lines, separated by a

CR-LF pair and terminated by an extra CR-LF pair. These lines should be treated as the response to an HTTP Upgrade request for the WebSocket URI, For example:

```
HTTP/1.1 404 Not found

404 message body...
```

If F is not set, then the server has accepted the request. The following n bytes, where n is the value of len inside opcdata plus 1, are an 8-32 bit length of the request header data that follows, in network byte order. The request header data consists of a series of lines, separated by a CR-LF pair and terminated by an extra CR-LF pair. These are encoded according to enc as defined in the AddChannel request, and the complete set of headers after decoding is treated exactly as if it was received in response to a handshake on a separate connection.

2 - FlowControl

opcdata is defined as follows:

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

RSV is reserved for future use, and len is the number of bytes in 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?)

3-7 - 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 make a new WebSocket connection, it MAY choose to use an existing WebSocket connection if all of the following are true:

- the MUX 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 stream id

If the client chooses to reuse an existing MUXd connection, it sends an AddChannel message as described above. If the AddChannel is successful, 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 MUX blocks into larger WebSocket frames. 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 MUX implementation MUST ensure reasonable fairness among the logical channels. This is accomplished in several ways:

- by restricting the send quota of a logical channel, the receiver can make sure that sender cannot dominate its buffer space
- when sending data, 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.
- 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 mux/demux 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 MUX 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 MUX'd channels, or should we require that an intermediary MUXing 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. Security Considerations

TBD

15. 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

16. 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