HyBi Working Group
Internet-Draft
Intended status: Standards Track
Expires: January 3, 2014

J. Tamplin
T. Yoshino
Google, Inc.
July 2, 2013

A Multiplexing Extension for WebSockets
draft-ietf-hybi-websocket-multiplexing-11

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

Copyright Notice

Table of Contents

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 messages 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 a server supports this extension and configuration offered by the extension parameters in the peer client’s request, the server accepts the use of this extension by including a response in 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 integer ID for each multiplexed connection. Each logical channel with a non-zero integer ID exchanges frames of the multiplexed connection of that ID. The logical channel with an ID of 0 exchanges data to control multiplexing.

The ID used for distinguishing data for different logical channels is attached to each encapsulated frames. It is placed at the head of a message that encapsulates the original frame of a multiplex connection. The field is called "logical channel ID tag field".

1.2. Usecase

Multiplexing could be done by using Web Workers. One limitation of Web Workers is that it cannot multiplex traffic for different origins. WebSocket protocol level multiplexing enables that. Large scale service provides may employ layer-7 load balancing system. For such system, it’s good that multiplexing is specified at the protocol level, not application level.

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. Multiplexed Connections

This 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 client’s and server’s opening handshake of the physical connection are automatically taken to mean ones for the multiplexed connection after removing physical connection specific header entries. This automatically opened multiplexed connection is 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 message. Note that only the client may initiate new WebSocket connections. An AddChannelRequest contains any handshake headers for the corresponding new multiplexed connection. The server’s AddChannelResponse likewise contains any handshake headers for the corresponding new multiplexed connection.

Logical channel with an ID of 0 is reserved and called "control channel". It’s automatically opened for exchanging multiplex control messages.

If there’re existing connections between a client and a peer server where this multiplexing extensions was successfully negotiated, and the client wants to create a new logical channel, the client chooses one from them and add a new logical channel to the connection. Otherwise, the client SHOULD attempt to open a new underlying connection to the server and open a new WebSocket connection on it.

Once the multiplexing extension is negotiated on a connection, all frames of multiplexed connection are tagged with a channel ID number and encapsulated into binary messages. Channel IDs are assigned by the client on issuing an AddChannelRequest.

A receiver MAY process frames of different multiplexed connections in parallel. A receiver MUST process multiplex control messages exclusively.

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

If the tuple of /secure/ flag, /port/ [RFC6455] and the IP address which /host/ [RFC6455] resolves to is the same for multiple application instances, their connections may be multiplexed onto the

same physical connection by creating logical channels for each of the instances.

A logical channel with /secure/ flag [RFC6455] and one without /secure/ flag MUST NOT be multiplexed onto the same physical connection. An endpoint may be required to open another physical connection for this case even if there’s an existing physical connection with multiplexing extension successfully negotiated. For example, if a client is configured to use different TLS client certificates for each logical channel, the client needs to establish separate TLS connections.

4. Extension Negotiation

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

To request use of the WebSocket Multiplexing Extension, a client includes an element with the "mux" extension token as its extension identifier in the "Sec-WebSocket-Extensions" header in the client’s 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".

A server accepts use of the WebSocket Multiplexing Extension by including an element with the "mux" extension token in the "Sec-WebSocket-Extensions" header in the server’s opening handshake. The element has no extension parameter.

A server rejects use of the WebSocket Multiplexing Extension by not including the element for the extension in the "Sec-WebSocket-Extensions" header in the server’s 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.

5. 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, on the sender side that extension is applied to frames after multiplexing, and on the receiver side that extension is applied to frames before demultiplexing, unless otherwise noted in the extension’s spec.

A client MAY request an extension for both the physical connection and the "Implicitly Opened Connection" by placing extension entries before and after the entry of this multiplexing extension. If enabling the extension for both the physical connection and "Implicitly Opened Connection" doesn’t make sense, the server rejects either of them.

For example, if we have an extension called foobar that can be used either the physical connection or multiplexed connections, 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 for the _Implicitly Opened Connection_, or

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

to apply the foobar extension to the physical connection.

5.1. Ordering Extensions

5.1.1. 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 a 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. For intermediaries that want to demultiplex a connection with this extension and forward encapsulating messages to different backends, it’s also useful because each encapsulating message can be forwarded without uncompressing. However, compressing each multiplexed connection is expensive in terms of memory consumption.

5.1.2. 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 in which the compression extension is applied to the physical connection side.

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 message 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, a client picks 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 until a slot becomes available. An endpoint MUST _Fail the Logical Channel_ with drop reason code of 2007 when it’s clear that the other peer violates this rule about new channel slots.

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, a 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 logical channel is initialized when it sends its opening handshake for the physical connection. The "quota" extension parameter included in the extension offer for this multiplexing extension in the client’s opening handshake for the physical connection specifies the initial value of the server’s send quota. If the "quota" extension parameter is not specified, the initial value is set to 0. If the "quota" extension parameter is specified, the initial value is the parameter’s value parsed 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 that AddChannelRequest. Initialization timing is when the client completes sending the AddChannelRequest.

For a logical channel added by accepting an AddChannelRequest, a server gets "send quota" of 0. Initialization timing is when the

server completes 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 that logical 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 rule about send quota.

When a frame is sent on a logical channel with non-zero ID, the cost of the frame is subtracted from the "send quota" of the endpoint for that logical 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 unless the size of quota it can replenish is too small and therefore replenishing it pushes down overall performance.

7. Framing

The multiplexing extension uses binary messages to transfer both data for controlling multiplexing and data of multiplexed connections. These binary messages are called "encapsulating messages" and 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 messages 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
 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)                   |
+-+-+-+-------------------------------------------------------+
```

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

A field for which it’s specified to use 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 the meaning of encapsulating messages with data opcodes other than binary, endpoints MUST NOT send any data message other than "binary". An endpoint received such a message MUST _Fail the Physical Connection_ with drop

reason code of 2001.

An endpoint received a binary message with an incomplete or invalid logical channel ID tag field at the head of the message MUST _Fail the Physical Connection_ with drop reason code of 2002.

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

8. Encapsulation

This extension encapsulates each frame of a multiplexed connection into an encapsulating message. Payload Data of an encapsulating message is 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.

```
 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
+-----------------------------------------------------------------+
|                   Logical channel ID tag                        |
|                        (8/16/24/32)                             |
+ - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - +
|             Logical channel ID tag continued                    |
+-+-+-+-+-------+-------------------------------------------------+
|F|R|R|R| opcode|         Payload Data of original frame          |
|I|S|S|S|  (4)  |                                                 |
|N|V|V|V|       |                                                 |
| |1|2|3|       |                                                 |
+-+-+-+-+-------+ - - - - - - - - - - - - - - - - - - - - - - - - - +
:         Payload Data of original frame continued ...            :
+ - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - +
:         Payload Data of original frame continued ...            :
+-----------------------------------------------------------------+
```

A receiver restores the original frame from the Payload Data and deliver the restored frame 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 messages or adjust the amount of data to flush along with flow control.

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

When an encapsulated frame with the continuation opcode is received though there’s no preceding encapsulated 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 are 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. On the same logical channel, fragments for any other message MUST NOT be injected between fragments of a control message. A demultiplexer received an encapsulated frame with a control opcode and the FIN bit unset MUST process the following encapsulated frames on the same logical channel as the encapsulated fragments of that control message until it encounters one with the FIN bit set. A demultiplexer encountered any encapsulated frame whose opcode is not continuation injected between fragments of a control message on the same logical channel 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 don’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 Messages

A binary message with the logical channel ID of 0 contains one multiplex control block in the "Payload data" portion.

```
 0 1 2 3 4 5 6 7
+---------------+
|Channel ID of 0|
+---------------+
|Multiplex      |
:control block  :
|               |
+---------------+
```

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 message 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.

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 0x7FFFFFFFFFFFFFFF 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.

- o The most significant bit of the first octet MUST be 0.
- o The minimal number of bytes necessary to represent the integer MUST be used.
- o 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   |
+-+-+-+---------+
|Objective      |
:channel ID     :
|(1-4 octet)    |
+---------------+
|Handshake      |
:               :
|               |
+---------------+
```

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

The rest is the handshake field. The client’s opening handshake as defined in Section 4 of RFC 6455 [RFC6455] for the new multiplexed connection including the CRLF following the last header. The "Upgrade", "Sec-WebSocket-Key" and "Sec-WebSocket-Version" header are excluded. The "Sec-WebSocket-Extensions" header contains only extensions applied to the multiplexed connection. 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 discarded by the peer server. 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.

Multiplex control opcode of the AddChannelResponse is 1.

AddChannelResponse has fields as follows:

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

F

Failure bit.

If the failure bit is not set, then the server has accepted the AddChannelRequest. The handshake field contains 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 contains 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.

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

The rest is the handshake field. The server's opening handshake as defined in Section 4 of RFC 6455 [RFC6455] for this multiplexed connection. The "Upgrade" and "Sec-WebSocket-Accept" header are excluded. The "Sec-WebSocket-Extensions" header contains only extensions applied to the multiplexed connection. 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 this 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         |
:               :
|               |
+---------------+
```

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 rest is 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.

--- To be fixed ---

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 and RSV field. Objective channel ID is 2.

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:

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

If a client chooses to share the existing WebSocket connection with multiplexing, it sends an AddChannelRequest as described above. If an 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

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 among Logical Channels

A multiplexing implementation may be requested to 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.

- Determine send quota for a logical channel 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

- Use a fair algorithm to select 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.

- 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.

18. 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 5.1.2, 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

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

[RFC2818] Rescorla, E., "HTTP Over TLS", RFC 2818, May 2000.

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

22.2. Informative References

[CRIME] Rizzo, J. and T. Duong, "The CRIME attack", Ekoparty 2012, September 2012.

Authors' Addresses

John A. Tamplin
Google, Inc.

Email: jat@jaet.org

Takeshi Yoshino
Google, Inc.

Email: tyoshino@google.com

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

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

July 17, 2012

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

Abstract

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

Status of this Memo

Copyright Notice

Table of Contents

1. Introduction

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

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

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

1.1. Idle Connection Timeouts and Connection Reuse

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

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

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

1.2. Terminology

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

2. Keep-Alive Header

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

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

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

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

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

2.1. ’timeout’ Parameter

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

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

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

Each peer, client or server, has a different view of the time that a connection becomes idle. Packet transmission at one peer necessarily occurs before receipt, meaning that the sending peer perceives the connection as being idle earlier to the receiving peer. Similarly, the buffering or retransmission of data by lower layers of the stack,

which is unlikely to be visible to the HTTP implementation, compounds this effect. Clients are advised to make allowances for delays in determining whether to reuse an idle connection.

2.2. Other Header Parameters

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

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

2.2.1. 'max' Parameter

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

3. Existing Intermediaries

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

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

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

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

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

4. Upgraded HTTP Connections

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

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

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

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

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

5. Examples

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

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

Figure 1: Independent HTTP Hops

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

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

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

Figure 2: Interdependent Connections with Upgrade

6. Security Considerations

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

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

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

7. IANA Considerations

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

7.1. Registration for Keep-Alive Header

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

Header field: Keep-Alive

Applicable protocol: HTTP

Status: standard

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

Specification document(s): RFCXXXX (this document)

7.2. Registry for Keep-Alive Information

This document establishes a registry for Keep-Alive Information.

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

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

The registry includes the following initial values:

timeout See Section 2.1 of this document.

max Deprecated. See Section 2.2.1 of this document.

8. Acknowledgments

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

9. Change Log

Since -01:

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

Since -00:

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

Since draft-loreto-http-timeout:

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

10. References

10.1. Normative References

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

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

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

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

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

10.2. Informative References

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

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

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

Authors' Addresses

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

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

Salvatore Loreto
Ericsson
Hirsalantie 11
Jorvas 02420
Finland

Email: salvatore.loreto@ericsson.com

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

Email: gregw@intalio.com