HyBi Working Group
Internet-Draft
Intended status: Standards Track
Expires: May 16, 2015

T. Yoshino
Google, Inc.

November 12, 2014

Compression Extensions for WebSocket
draft-ietf-hybi-permessage-compression-19

Abstract

This document specifies a framework for creating WebSocket extensions that add compression functionality to the WebSocket Protocol. An extension based on this framework compresses the payload data portion of non-control WebSocket messages on a per-message basis using parameters negotiated during the opening handshake. This framework provides a general method to apply a compression algorithm to the contents of WebSocket messages. For each compression algorithm, an extension is defined by specifying parameter negotiation and payload transformation algorithm in detail. This document also specifies one specific compression extension using the DEFLATE algorithm.

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

Status of this Memo

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

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

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

This Internet-Draft will expire on May 16, 2015.

Copyright Notice

Table of Contents

1. Introduction

This document specifies a framework to add compression functionality to the WebSocket Protocol [RFC6455]. This framework specifies how to define WebSocket Per-message Compression Extensions (PMCEs) individually for various compression algorithms based on the extension concept of the WebSocket Protocol specified in Section 9 of [RFC6455]. A WebSocket client and a peer WebSocket server negotiate use of a PMCE and determine parameters to configure the compression algorithm during the WebSocket opening handshake. The client and server can then exchange non-control messages using frames with compressed data in the payload data portion.

This framework only specifies a general method to apply a compression algorithm to the contents of WebSocket messages. A document specifying an individual PMCE describes how to negotiate configuration parameters for the compression algorithm and how to transform (compress and decompress) data in the payload data portion in detail.

A WebSocket client may offer multiple PMCEs during the WebSocket opening handshake. A peer WebSocket server received those offers may choose and accept preferred one or decline all of them. PMCEs use the RSV1 bit of the WebSocket frame header to indicate whether a message is compressed or not, so that an endpoint can choose not to compress messages with incompressible contents.

This document also specifies one specific PMCE based on the DEFLATE [RFC1951] algorithm. The extension name of the PMCE is "permessage-deflate". We chose DEFLATE since it's widely available as a library on various platforms and the overhead is small. To align the end of compressed data to an octet boundary, this extension uses the algorithm described in Section 2.1 of [RFC1979]. Endpoints can take over the LZ77 sliding window [LZ77] used to build frames for previous messages to get better compression ratio. For resource-limited devices, this extension provides parameters to limit memory usage for

compression context.

2. Conformance Requirements and Terminology

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].

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 can 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 understand and are not intended to be performant.

This document references the procedure to _Fail the WebSocket Connection_. This procedure is defined in Section 7.1.7 of [RFC6455].

This document references the event that _The WebSocket Connection is Established_ and the event that _A WebSocket Message Has Been Received_. These events are defined in Section 4.1 and Section 6.2, respectively, of [RFC6455].

This document uses the Augmented Backus-Naur Form (ABNF) notation of [RFC5234]. The DIGIT (decimal 0-9) rule is included by reference, as defined in the Appendix B.1 of [RFC5234].

3. Complementary Terminology

This document defines some terms about WebSocket and WebSocket Extension mechanism that are underspecified or not defined at all in [RFC6455]. This terminology is effective only in this document and any other documents that refer to this section.

"A non-control message" means a message that consists of non-control frames as defined in Section 5.6 of [RFC6455].

"A message payload (or payload of a message)" means concatenation of the payload data portion of all frames representing a single message, as well as how /data/ is formed from in Section 6.2 of [RFC6455].

"An extension in use next to extension X" means the extension listed next to X in the "Sec-WebSocket-Extensions" header in the server's opening handshake as defined in Section 9.1 of [RFC6455]. Such an extension is applied to outgoing data from the application right after X on sender side but applied right before X to incoming data from the underlying transport.

"An extension in use preceding extension X" means the extension

listed right before X in the "Sec-WebSocket-Extensions" header in the server's opening handshake. Such an extension is applied to outgoing data from the application right before X on sender side but applied right after X to incoming data from the underlying transport.

"An extension negotiation offer" means each element in the "Sec-WebSocket-Extensions" header in the client's opening handshake.

"An extension negotiation response" means each element in the "Sec-WebSocket-Extensions" header in the server's opening handshake.

"A corresponding extension negotiation response for an extension negotiation offer" means an extension negotiation response a server sends back to the peer client that contains the same extension name as the offer and meets the requirements represented by the offer.

"Accepting an extension negotiation offer" means including a corresponding extension negotiation response for the offer in the "Sec-WebSocket-Extensions" header in the server's opening handshake.

"Declining an extension negotiation offer" means not including a corresponding extension negotiation response for the offer in the "Sec-WebSocket-Extensions" header in the server's opening handshake.

4. WebSocket Per-message Compression Extension

WebSocket Per-message Compression Extensions (PMCEs) are extensions to the WebSocket Protocol enabling compression functionality. PMCEs are built based on the extension concept of the WebSocket Protocol specified in Section 9 of [RFC6455]. PMCEs are individually defined for each compression algorithm to be implemented, and are registered in the WebSocket Extension Name Registry created in Section 11.4 of [RFC6455]. Each PMCE refers to this framework and defines the following:

- The contents of its extension negotiation offer/response to include in the "Sec-WebSocket-Extensions" header. The contents include the extension name of the PMCE and any applicable extension parameters.

- How to interpret the extension parameters exchanged during the opening handshake

- How to transform the payload of a message.

One such extension is defined in Section 8 of this document and is registered in Section 10. Other PMCEs may be defined in other documents.

Section 5 describes the basic extension negotiation process. Section 6 describes how to apply the compression algorithm with negotiated parameters to the contents of WebSocket messages.

5. Extension Negotiation

To offer use of a PMCE, a client includes a "Sec-WebSocket-Extensions" header element with the extension name of the PMCE in the "Sec-WebSocket-Extensions" header in the client's opening handshake of the WebSocket connection. Extension parameters in the element represent the PMCE offer in detail. For example, a client lists preferred configuration parameter values for the

compression algorithm of the PMCE. A client offers multiple PMCE choices to the server by including multiple elements in the "Sec-WebSocket-Extensions" header, one for each PMCE offered. The set of elements MAY include multiple PMCEs with the same extension name to offer use of the same algorithm with different configuration parameters. The order of elements means the client's preference. An element precedes another element has higher preference. It is RECOMMENDED that a server accepts PMCEs with higher preference if the server supports it.

A PMCE negotiation offer informs requests and/or hints to the server. A request in a PMCE negotiation offer indicates constraints on the server's behavior that must be satisfied if the server accepts the offer. A hint in a PMCE negotiation offer indicates information about the client's behavior that the server may either safely ignore or refer to when the server decides its behavior.

To accept use of an offered PMCE, a server includes a "Sec-WebSocket-Extensions" header element with the extension name of the PMCE in the "Sec-WebSocket-Extensions" header in the server's opening handshake of the WebSocket connection. Extension parameters in the element represent the configuration parameters of the PMCE to use in detail. We call these extension parameters and their values "agreed parameters". The element MUST represent a PMCE that is fully supported by the server. The contents of the element doesn't need to be exactly the same as one of the received extension negotiation offers. For example, an extension negotiation offer with an extension parameter "X" indicating availability of the feature X may be accepted with an element without the extension parameter meaning that the server declined use of the feature.

"Agreed parameters" MUST represent how the requests and hints in the client's extension negotiation offer have been handled in addition to the server's requests and hints on the client's behavior, so that the client can configure its behavior without identifying which PMCE extension negotiation offer has been accepted.

For example, if a client sends an extension negotiation offer including a parameter "enable_compression" and another without the parameter, the server accepts the former and tell that to the client

by sending back an element including a parameter that acknowledges

"enable_compression". The name of the acknowledging parameter doesn't need to be the same as the offer.

General negotiation flow will be like the following. How to handle parameters in detail will be specified in the specifications for each PMCE.

A client makes an offer including parameters identifying the following:

- Hints about how the client is planning to compress data
- Requests about how the server compresses data
- Limitation of the client's compression functionality

The peer server makes a determination of its behavior based on these parameters if it can and wants to proceed with this PMCE enabled, and responds to the client with parameters identifying the following:

- Requests about how the client compresses data
- How the server will compress data

The client makes a determination of its behavior based on these parameters from the server if it can and wants to proceed with this PMCE enabled. Otherwise, the client starts closing handshake with close code 1010.

There can be compression features that can be applied separately for each direction. For such features, the acknowledging parameter and the parameter for the reverse direction must be chosen to be distinguishable from each other. For example, we can add "server_" prefix to parameters affecting data sent from a server and "client_" prefix to ones affecting data sent from a client to make them distinguishable.

A server MUST NOT accept a PMCE extension negotiation offer together with another extension if the PMCE will conflict with the extension on use of the RSV1 bit. A client that received a response accepting a PMCE extension negotiation offer together with such an extension MUST _Fail the WebSocket Connection_.

A server MUST NOT accept a PMCE extension negotiation offer together with another extension if the PMCE will be applied to output of the extension and any of the following conditions applies to the extension:

- The extension requires boundary of fragments to be preserved between output from the extension at the sender and input to the extension at the receiver.

- The extension uses the "Extension data" field or any of the reserved bits on the WebSocket header as a per-frame attribute.

A client that received a response accepting a PMCE extension negotiation offer together with such an extension MUST _Fail the WebSocket Connection_.

A server declines all offered PMCEs by not including any element with PMCE names. If a server responds with no PMCE element in the "Sec-WebSocket-Extensions" header, both endpoints proceed without Per-message Compression once _the WebSocket Connection is established_.

If a server gives an invalid response, such as accepting a PMCE that the client did not offer, the client MUST _Fail the WebSocket Connection_.

If a server responds with a valid PMCE element in the "Sec-WebSocket-Extensions" header and _the WebSocket Connection is established_, both endpoints MUST use the algorithm described in Section 6 and the message payload transformation (compressing and decompressing) procedure of the PMCE configured with the "agreed parameters" returned by the server to exchange messages.

5.1. Negotiation Examples

The following are example values for the "Sec-WebSocket-Extensions" header offering PMCEs. permessage-foo and permessage-bar in the examples are hypothetical extension names of PMCEs for compression algorithm foo and bar.

- Offer the permessage-foo.

```
permessage-foo
```

- Offer the permessage-foo with a parameter x with a value of 10.

```
permessage-foo; x=10
```

The value MAY be quoted.

```
permessage-foo; x="10"
```

- Offer the permessage-foo as first choice and the permessage-bar as a fallback plan.

  permessage-foo, permessage-bar

- Offer the permessage-foo with a parameter use_y which enables a feature y as first choice, and the permessage-foo without the use_y parameter as a fallback plan.

  permessage-foo; use_y, permessage-foo

6. Framing

PMCEs operate only on non-control messages.

This document allocates the RSV1 bit of the WebSocket header for PMCEs, and calls the bit the "Per-message Compressed" bit. On a WebSocket connection where a PMCE is in use, this bit indicates whether a message is compressed or not.

A message with the "Per-message Compressed" bit set on the first fragment of the message is called a "compressed message". Frames of a compressed message have compressed data in the payload data portion. An endpoint received a compressed message decompresses the concatenation of the compressed data of the frames of the message by following the decompression procedure specified by the PMCE in use. The endpoint uses the bytes corresponding to the application data portion in this decompressed data for the _A WebSocket Message Has Been Received_ event instead of the received data as-is.

A message with the "Per-message Compressed" bit unset on the first fragment of the message is called an "uncompressed message". Frames of an uncompressed message have uncompressed original data as-is in the payload data portion. An endpoint received an uncompressed message uses the concatenation of the application data portion of the frames of the message as-is for the _A WebSocket Message Has Been Received_ event.

6.1. Compression

An endpoint MUST use the following algorithm to send a message in the form of a compressed message.

1. Compress the message payload of the original message by following the compression procedure of the PMCE. The original message may be input from the application layer or output of another WebSocket extension depending on what extensions were negotiated.

2. If this PMCE is the last extension to process outgoing messages, build frame(s) by using the compressed data instead of the original data for the message payload, and setting the "Per-message Compressed" bit of the first frame, then send the frame(s) as described in Section 6.1 of RFC6455. Otherwise, pass the transformed message payload and modified header values including "Per-message Compressed" bit value set to 1 to the extension next to the PMCE. If the extension expects frames for input, build a frame for the message and pass it.

An endpoint MUST use the following algorithm to send a message in the

form of an uncompressed message. If this PMCE is the last extension to process outgoing messages, build frame(s) by using the original data for the payload data portion as-is and unsetting the "Per-message Compressed" bit of the first frame, then send the frame(s) as described in Section 6.1 of RFC6455. Otherwise, pass the message payload and header values to the extension next to the PMCE as-is. If the extension expects frames for input, build a frame for the message and pass it.

An endpoint MUST NOT set the "Per-message Compressed" bit of control frames and non-first fragments of a data message. An endpoint received such a frame MUST _Fail the WebSocket Connection_.

PMCEs don't change the opcode field. The opcode of the first frame of a compress message indicates the opcode of the original message.

The payload data portion in frames generated by a PMCE is not subject to the constraints for the original data type. For example, the concatenation of the output data corresponding to the application data portion of frames of a compressed text message is not required to be valid UTF-8. At the receiver, the payload data portion after decompression is subject to the constraints for the original data type again.

6.2. Decompression

An endpoint MUST use the following algorithm to receive a message in the form of a compressed message.

1. Concatenate the payload data portion of the received frames of the compressed message. The received frames may be direct input from the underlying transport or output of another WebSocket extension depending on what extensions were negotiated.

2. Decompress the concatenation by following the decompression procedure of the PMCE.

3. If this is the last extension to process incoming messages, deliver the _A WebSocket Message Has Been Received_ event to the application layer with the decompressed message payload and header values including the "Per-message Compressed" bit unset to 0. Otherwise, pass the decompressed message payload and header values including the "Per-message Compressed" bit unset to 0 to the extension preceding the PMCE. If the extension expects frames for input, build a frame for the message and pass it.

An endpoint MUST use the following algorithm to receive a message in the form of an uncompressed message. If this PMCE is the last

extension to process incoming messages, deliver the _A WebSocket Message Has Been Received_ event to the application layer with the received message payload and header values as-is. Otherwise, pass the message payload and header values to the extension preceding the PMCE as-is. If the extension expects frames for input, build a frame for the message and pass it.

7. Intermediaries

When an intermediary proxies a WebSocket connection, the intermediary MAY add, change or remove Per-message Compression on proxied messages if the intermediary meets all of the following requirements:

- The intermediary understands that Per-message Compression.
- The intermediary can read all data of the proxied WebSocket connection including the opening handshake request, opening handshake response, and messages.

- The intermediary can alter the proxied data before forwarding them accordingly to conform to the constraints of the new combination of extensions. For example, if Per-message Compression is removed from messages, the corresponding element in the "Sec-WebSocket-Extensions" in the opening handshake response which enabled the Per-message Compression must also be removed.

Otherwise, the intermediary MUST NOT add, change or remove Per-message Compression on proxied messages.

8. permessage-deflate extension

This section specifies a specific PMCE called "permessage-deflate". It compresses the payload of a message using the DEFLATE algorithm

[RFC1951] and the byte boundary aligning method introduced in [RFC1979].

This section uses the term "byte" with the same meaning as RFC1951, i.e. 8 bits stored or transmitted as a unit (same as an octet).

The registered extension name for this extension is "permessage-deflate".

Four extension parameters are defined for permessage-deflate to help endpoints manage per-connection resource usage.

- "server_no_context_takeover"
- "client_no_context_takeover"
- "server_max_window_bits"
- "client_max_window_bits"

These parameters enable two methods (no_context_takeover and max_window_bits) of constraining memory usage that may be applied independently to either direction of WebSocket traffic. The extension parameters with the "client_" prefix are used to negotiate DEFLATE parameters to control compression on messages sent by a client and received by a server. The client refers to parameters with the "client_" prefix to configure its compressor, while the server refers to them to configure its decompressor. The extension parameters with the "server_" prefix are used to negotiate DEFLATE parameters to control compression on messages sent by a server and received by a client. The server refers to parameters with the "server_" prefix to configure its compressor, while the client refers to them to configure its decompressor. All of these four parameters are defined for both a client's extension negotiation offer and a server's extension negotiation response.

A server MUST decline an extension negotiation offer for this extension if any of the following conditions is met:

- The offer has any extension parameter not defined for use in an offer.
- The offer has any extension parameter with an invalid value.

- The offer has multiple extension parameters with the same name.

- The server doesn't support the offered configuration.

A client MUST _Fail the WebSocket Connection_ if the peer server accepted an extension negotiation offer for this extension with an extension negotiation response meeting any of the following condition:

- The response has any extension parameter not defined for use in a response.

- The response has any extension parameter with an invalid value.

- The response has multiple extension parameters with the same name.

- The client doesn't support the configuration that the response represents.

The term "LZ77 sliding window" used in this section means the buffer used by the DEFLATE algorithm to store recently processed input. The DEFLATE compression algorithm searches the buffer for match with the next input.

The term "use context take over" used in this section means to use the same LZ77 sliding window the endpoint used to build frames of the last sent message to build frames of the next message.

8.1. Method Parameters

8.1.1. Context Takeover Control

8.1.1.1. server_no_context_takeover

A client MAY include the "server_no_context_takeover" extension parameter in an extension negotiation offer. This extension parameter has no value. By including this extension parameter in an extension negotiation offer, a client prevents the peer server from using context take over. If the peer server doesn't use context take over, the client doesn't need to reserve memory to retain the LZ77 sliding window in between messages.

Absence of this extension parameter in an extension negotiation offer indicates that the client can receive a message which the server built using context take over.

A server accepts an extension negotiation offer including the

"server_no_context_takeover" extension parameter by including the

"server_no_context_takeover" extension parameter in the corresponding extension negotiation response to send back to the client. The "server_no_context_takeover" extension parameter in an extension negotiation response has no value.

It is RECOMMENDED that a server supports the "server_no_context_takeover" extension parameter in an extension negotiation offer.

A server MAY include the "server_no_context_takeover" extension parameter in an extension negotiation response even if the extension negotiation offer being accepted by the extension negotiation response didn't have the "server_no_context_takeover" extension parameter.

8.1.1.2. client_no_context_takeover

A client MAY include the "client_no_context_takeover" extension parameter in an extension negotiation offer. This extension parameter has no value. By including this extension parameter in an extension negotiation offer, a client informs the peer server of a hint that even if the server won't include the "client_no_context_takeover" extension parameter in the corresponding extension negotiation response to the offer, the client is not going to use context take over.

A server MAY include the "client_no_context_takeover" extension parameter in an extension negotiation response. If the received extension negotiation offer includes the "client_no_context_takeover" extension parameter, the server may either ignore the parameter or use the parameter to avoid taking over an LZ77 sliding window unnecessarily by including "client_no_context_takeover" extension parameter in the corresponding extension negotiation response to the offer. The "client_no_context_takeover" extension parameter in an extension negotiation response has no value. By including the "client_no_context_takeover" extension parameter in an extension negotiation response, a server prevents the peer client from using context take over. This reduces the amount of memory that the server has to reserve for the connection.

Absence of this extension parameter in an extension negotiation response indicates that the server can receive messages built by the client using context take over.

A client MUST support the "client_no_context_takeover" extension parameter in an extension negotiation response.

8.1.2. Limiting the LZ77 sliding window size

8.1.2.1. server_max_window_bits

A client MAY include the "server_max_window_bits" extension parameter in an extension negotiation offer. This parameter has a decimal integer value without leading zeroes between 8 to 15 inclusive indicating the base-2 logarithm of the LZ77 sliding window size and MUST conform to the ABNF below.

```
server_max_window_bits = 1*DIGIT
```

By including this parameter in an extension negotiation offer, a client limits the LZ77 sliding window size that the server uses to compress messages. If the peer server uses small LZ77 sliding window to compress messages, the client can reduce the memory for the LZ77 sliding window.

A server declines an extension negotiation offer with this parameter if the server doesn't support it.

Absence of this parameter in an extension negotiation offer indicates that the client can receive messages compressed using an LZ77 sliding window of up to 32,768 bytes.

A server accepts an extension negotiation offer with this parameter by including the "server_max_window_bits" extension parameter in the extension negotiation response to send back to the client with the same or smaller value as the offer. The "server_max_window_bits" extension parameter in an extension negotiation response has a decimal integer value without leading zeroes between 8 to 15 inclusive indicating the base-2 logarithm of the LZ77 sliding window size and MUST conform to the ABNF below.

```
    server_max_window_bits = 1*DIGIT
```

A server MAY include the "server_max_window_bits" extension parameter in an extension negotiation response even if the extension negotiation offer being accepted by the response didn't include the "server_max_window_bits" extension parameter.

8.1.2.2. client_max_window_bits

A client MAY include the "client_max_window_bits" extension parameter in an extension negotiation offer. This parameter has no value or a decimal integer value without leading zeroes between 8 to 15 inclusive indicating the base-2 logarithm of the LZ77 sliding window size. If a value is specified for this parameter, the value MUST

conform to the ABNF below.

```
    client_max_window_bits = 1*DIGIT
```

By including this parameter in an offer, a client informs the peer server of that the client supports the "client_max_window_bits" extension parameter in an extension negotiation response, and optionally a hint by attaching a value to the parameter. If the "client_max_window_bits" extension parameter in an extension negotiation offer has a value, the parameter also informs the peer server of a hint that even if the server won't include the "client_max_window_bits" extension parameter in the corresponding extension negotiation response with a value greater than one in the extension negotiation offer or the server doesn't include the extension parameter at all, the client is not going to use LZ77 sliding window size greater than the size specified by the value in the extension negotiation offer to compress messages.

If a received extension negotiation offer has the "client_max_window_bits" extension parameter, the server MAY include the "client_max_window_bits" extension parameter in the corresponding extension negotiation response to the offer. If the "client_max_window_bits" extension parameter in a received extension negotiation offer has a value, the server may either ignore this value or use this value to avoid allocating an unnecessarily big LZ77 sliding window by including the "client_max_window_bits" extension

parameter in the corresponding extension negotiation response to the offer with a value equal to or smaller than the received value. The "client_max_window_bits" extension parameter in an extension negotiation response has a decimal integer value without leading zeroes between 8 to 15 inclusive indicating the base-2 logarithm of the LZ77 sliding window size and MUST conform to the ABNF below.

```
client_max_window_bits = 1*DIGIT
```

By including this extension parameter in an extension negotiation response, a server limits the LZ77 sliding window size that the client uses to compress messages. This reduces the amount of memory for decompression context that the server has to reserve for the connection.

If a received extension negotiation offer doesn't have the "client_max_window_bits" extension parameter, the corresponding extension negotiation response to the offer MUST NOT include the "client_max_window_bits" extension parameter.

Absence of this extension parameter in an extension negotiation response indicates that the server can receive messages compressed

using an LZ77 sliding window of up to 32,768 bytes.

8.1.3. Example

The simplest "Sec-WebSocket-Extensions" header in a client's opening handshake to offer use of the permessage-deflate is as follows:

```
Sec-WebSocket-Extensions: permessage-deflate
```

Since the "client_max_window_bits" extension parameter is not included in this extension negotiation offer, the server must not accept the offer with an extension negotiation response including the "client_max_window_bits" extension parameter. The simplest "Sec-WebSocket-Extensions" header in a server's opening handshake to accept use of the permessage-deflate is the same.

The following extension negotiation offer sent by a client is asking the server to use the LZ77 sliding window size of 1,024 bytes or less and declaring that the client supports the "client_max_window_bits"

extension parameter in an extension negotiation response.

```
Sec-WebSocket-Extensions:
    permessage-deflate;
    client_max_window_bits; server_max_window_bits=10
```

This extension negotiation offer might be rejected by the server because the server doesn't support the "server_max_window_bits" extension parameter in an extension negotiation offer. This is fine if the client cannot receive messages compressed using a larger sliding window size, but if the client just prefers using a small window but wants to fallback to the "permessage-deflate" without the "server_max_window_bits" extension parameter, the client can make an offer with the fallback option like this:

```
Sec-WebSocket-Extensions:
    permessage-deflate;
    client_max_window_bits; server_max_window_bits=10,
    permessage-deflate;
    client_max_window_bits
```

The server can accept permessage-deflate by picking the supported one from the listed offers. To accept the first option, for example, the server may send back a response as follows:

```
Sec-WebSocket-Extensions:
    permessage-deflate; server_max_window_bits=10
```

To accept the second option, for example, the server may send back a

response as follows:

```
Sec-WebSocket-Extensions: permessage-deflate
```

8.2. Message Payload Transformation

8.2.1. Compression

An endpoint uses the following algorithm to compress a message.

1. Compress all the octets of the payload of the message using DEFLATE.

2. If the resulting data does not end with an empty DEFLATE block with no compression (the "BTYPE" bits are set to 00), append an empty DEFLATE block with no compression to the tail end.

3. Remove 4 octets (that are 0x00 0x00 0xff 0xff) from the tail end. After this step, the last octet of the compressed data contains (possibly part of) the DEFLATE header bits with the "BTYPE" bits set to 00.

When using DEFLATE in the first step above:

- An endpoint MAY use multiple DEFLATE blocks to compress one message.
- An endpoint MAY use DEFLATE blocks of any type.
- An endpoint MAY use both DEFLATE blocks with the "BFINAL" bit set to 0 and DEFLATE blocks with the "BFINAL" bit set to 1.
- When any DEFLATE block with the "BFINAL" bit set to 1 doesn't end at a byte boundary, an endpoint MUST add minimal padding bits of 0 to make it end at a byte boundary. The next DEFLATE block follows the padded data if any.

An endpoint fragments a compressed message by splitting the result of running this algorithm. Even when only a part of payload is available, a fragment can be built by compressing the available data and choosing block type appropriately so that the end of the resulting compressed data is aligned at a byte boundary. Note that for non-final fragments, the removal of 0x00 0x00 0xff 0xff must not be done.

An endpoint MUST NOT use an LZ77 sliding window longer than 32,768 bytes to compress messages to send.

If the "agreed parameters" contain the "client_no_context_takeover" extension parameter, the client MUST start compressing each new message with an empty LZ77 sliding window. Otherwise, the client MAY take over the LZ77 sliding window used to build the last compressed message. Note that even if the client has included the

"client_no_context_takeover" extension parameter in its offer, the client MAY take over the LZ77 sliding window used to build the last compressed message if the "agreed parameters" don't contain the "client_no_context_takeover" extension parameter. The client-to-server "client_no_context_takeover" extension parameter is just a hint for the server to build an extension negotiation response.

If the "agreed parameters" contain the "server_no_context_takeover" extension parameter, the server MUST start compressing each new message with an empty LZ77 sliding window. Otherwise, the server MAY take over the LZ77 sliding window used to build the last compressed message.

If the "agreed parameters" contain the "client_max_window_bits" extension parameter with a value of w, the client MUST NOT use an LZ77 sliding window longer than the w-th power of 2 bytes to compress messages to send. Note that even if the client has included in its offer the "client_max_window_bits" extension parameter with a value smaller than one in the "agreed parameters", the client MAY use an LZ77 sliding window with any size to compress messages to send as long as the size conforms to the "agreed parameters". The client-to-server "client_max_window_bits" extension parameter is just a hint for the server to build an extension negotiation response.

If the "agreed parameters" contain the "server_max_window_bits" extension parameter with a value of w, the server MUST NOT use an LZ77 sliding window longer than the w-th power of 2 bytes to compress messages to send.

8.2.2. Decompression

An endpoint uses the following algorithm to decompress a message.

1. Append 4 octets of 0x00 0x00 0xff 0xff to the tail end of the payload of the message.

2. Decompress the resulting data using DEFLATE.

If the "agreed parameters" contain the "server_no_context_takeover" extension parameter, the client MAY decompress each new message with an empty LZ77 sliding window. Otherwise, the client MUST decompress each new message using the LZ77 sliding window used to process the last compressed message.

If the "agreed parameters" contain the "client_no_context_takeover" extension parameter, the server MAY decompress each new message with an empty LZ77 sliding window. Otherwise, the server MUST decompress each new message using the LZ77 sliding window used to process the last compressed message. Note that even if the client has included the "client_no_context_takeover" extension parameter in its offer, the server MUST decompress each new message using the LZ77 sliding window used to process the last compressed message if the "agreed parameters" don't contain the "client_no_context_takeover" extension parameter. The client-to-server "client_no_context_takeover" extension parameter is just a hint for the server to build an extension negotiation response.

If the "agreed parameters" contain the "server_max_window_bits" extension parameter with a value of w, the client MAY reduce the size of its LZ77 sliding window to decompress received messages down to the w-th power of 2 bytes. Otherwise, the client MUST use a 32,768 byte LZ77 sliding window to decompress received messages.

If the "agreed parameters" contain the "client_max_window_bits" extension parameter with a value of w, the server MAY reduce the size of its LZ77 sliding window to decompress received messages down to the w-th power of 2 bytes. Otherwise, the server MUST use a 32,768 byte LZ77 sliding window to decompress received messages. Note that even if the client has included in its offer the "client_max_window_bits" extension parameter with a value smaller than one in the "agreed parameters", the client MUST use an LZ77 sliding window of a size that conforms the "agreed parameters" to compress messages to send. The client-to-server "client_max_window_bits" extension parameter is just a hint for the server to build an extension negotiation response.

8.2.3. Examples

This section introduces examples of how the permessage-deflate transforms messages.

8.2.3.1. A message compressed using 1 compressed DEFLATE block

Suppose that an endpoint sends a text message "Hello". If the endpoint uses 1 compressed DEFLATE block (compressed with fixed Huffman code and the "BFINAL" bit is not set) to compress the message, the endpoint obtains the compressed data to use for the message payload as follows.

The endpoint compresses "Hello" into 1 compressed DEFLATE block and flushes the resulting data into a byte array using an empty DEFLATE block with no compression:

```
0xf2 0x48 0xcd 0xc9 0xc9 0x07 0x00 0x00 0x00 0xff 0xff
```

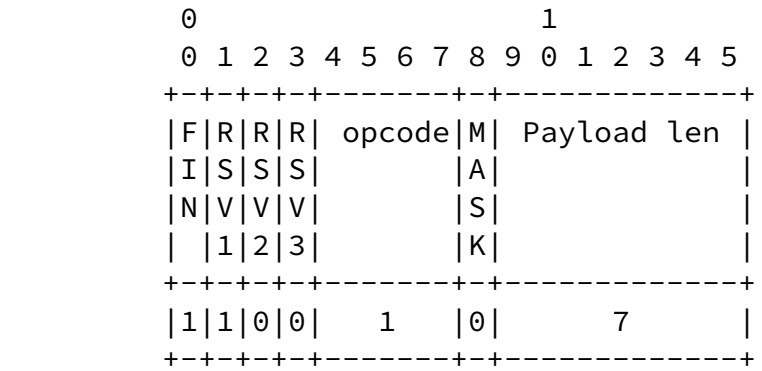
By stripping 0x00 0x00 0xff 0xff from the tail end, the endpoint gets the data to use for the message payload:

```
0xf2 0x48 0xcd 0xc9 0xc9 0x07 0x00
```

Suppose that the endpoint sends this compressed message without fragmentation. The endpoint builds one frame by putting the whole compressed data in the payload data portion of the frame:

```
0xc1 0x07 0xf2 0x48 0xcd 0xc9 0xc9 0x07 0x00
```

The first 2 octets (0xc1 0x07) are the WebSocket frame header (FIN=1, RSV1=1, RSV2=0, RSV3=0, opcode=text, MASK=0, Payload length=7). The following figure shows what value is set in each field of the WebSocket frame header.



Suppose that the endpoint sends the compressed message with fragmentation. The endpoint splits the compressed data into fragments and builds frames for each fragment. For example, if the fragments are 3 and 4 octet, the first frame is:

```
0x41 0x03 0xf2 0x48 0xcd
```

and the second frame is:

```
0x80 0x04 0xc9 0xc9 0x07 0x00
```

Note that the RSV1 bit is set only on the first frame.

8.2.3.2. Sharing LZ77 Sliding Window

Suppose that a client has sent a message "Hello" as a compressed message and will send the same message "Hello" again as a compressed message.


```
0xf2 0x48 0xcd 0xc9 0xc9 0x07 0x00
```

This is the payload of the first message the client has sent. If the "agreed parameters" contain the "client_no_context_takeover" extension parameter, the client compresses the payload of the next message into the same bytes (if the client uses the same "BTYPE" value and "BFINAL" value). So, the payload of the second message will be:

```
0xf2 0x48 0xcd 0xc9 0xc9 0x07 0x00
```

If the "agreed parameters" did not contain the "client_no_context_takeover" extension parameter, the client can compress the payload of the next message into shorter bytes by referencing the history in the LZ77 sliding window. So, the payload of the second message will be:

```
0xf2 0x00 0x11 0x00 0x00
```

So, 2 bytes are saved in total.

Note that even if some uncompressed messages (with the RSV1 bit unset) are inserted between the two "Hello" messages, they don't affect the LZ77 sliding window.

8.2.3.3. Using a DEFLATE Block with No Compression

```
0xc1 0x0b 0x00 0x05 0x00 0xfa 0xff 0x48 0x65 0x6c 0x6c 0x6f 0x00
```

This is a frame constituting a text message "Hello" built using a DEFLATE block with no compression. The first 2 octets (0xc1 0x0b) are the WebSocket frame header (FIN=1, RSV1=1, RSV2=0, RSV3=0, opcode=text, MASK=0, Payload length=7). Note that the RSV1 bit is set for this message (only on the first fragment if the message is

fragmented) because the RSV1 bit is set when DEFLATE is applied to the message, including the case when only DEFLATE blocks with no compression are used. The third to 13th octet consists a payload data containing "Hello" compressed using a DEFLATE block with no compression.

8.2.3.4. Using a DEFLATE Block with BFINAL Set to 1

On platform where the flush method using an empty DEFLATE block with no compression is not available, implementors can choose to flush data using DEFLATE blocks with "BFINAL" set to 1.

```
0xf3 0x48 0xcd 0xc9 0xc9 0x07 0x00 0x00
```


This is a payload of a message containing "Hello" compressed using a DEFLATE block with "BFINAL" set to 1. The first 7 octets constitute a DEFLATE block with "BFINAL" set to 1 and "BTYPE" set to 01 containing "Hello". The last 1 octet (0x00) contains the header bits with "BFINAL" set to 0 and "BTYPE" set to 00, and 5 padding bits of 0. This octet is necessary to allow the payload to be decompressed in the same manner as messages flushed using DEFLATE blocks with BFINAL unset.

8.2.3.5. Two DEFLATE Blocks in 1 Message

Two or more DEFLATE blocks may be used in 1 message.

```
0xf2 0x48 0x05 0x00 0x00 0x00 0xff 0xff 0xca 0xc9 0xc9 0x07 0x00
```

The first 3 octets (0xf2 0x48 0x05) and the least significant two bits of the 4th octet (0x00) constitute one DEFLATE block with "BFINAL" set to 0 and "BTYPE" set to 01 containing "He". The rest of the 4th octet contains the header bits with "BFINAL" set to 0 and "BTYPE" set to 00, and the 3 padding bits of 0. Together with the following 4 octets (0x00 0x00 0xff 0xff), the header bits constitute an empty DEFLATE block with no compression. A DEFLATE block containing "llo" follows the empty DEFLATE block.

8.2.3.6. Generating an Empty Fragment Manually

Suppose that an endpoint is sending data with unknown size. The endpoint may encounter the end of data signal from the data source

when its buffer for uncompressed data is empty. In such a case, the endpoint just needs to send the last fragment with FIN bit set to 1 and payload set to DEFLATE block(s) which contains 0 byte data. If the compression library being used doesn't generate any data when its buffer is empty, an empty uncompressed DEFLATE block can be built manually and used for this purpose as follows:

```
0x00
```

The only octet 0x00 contains the header bits with "BFINAL" set to 0 and "BTYPE" set to 00, and 5 padding bits of 0.

8.3. Implementation Notes

On most common software development platforms, their DEFLATE compression library provides a method to align compressed data to byte boundaries using an empty DEFLATE block with no compression. For example, Zlib [Zlib] does this when "Z_SYNC_FLUSH" is passed to the deflate function.

Some platforms may provide only methods to output and process compressed data with ZLIB header and Adler-32 checksum. On such platforms, developers need to write stub code to remove and complement them manually.

To obtain a useful compression ratio, an LZ77 sliding window size of 1,024 or more is RECOMMENDED.

On the direction where context takeover is disallowed, an endpoint can easily figure out whether a certain message will be shorter if compressed or not.. Otherwise, it's not easy to know whether future messages will benefit from having a certain message compressed. Implementor may employ some heuristics to determine this.

8.4. Intermediaries

When an intermediary forwards a message, the intermediary MAY change compression on the messages as far as the resulting sequence of messages conforms to the constraints based on the "agreed parameters". For example, an intermediary may decompress a received message, unset the "Per-message Compressed" bit and forward it to the

other peer. Since such a compression change may affect the LZ77 sliding window, the intermediary may need to parse and transform the following messages, too.

9. Security Considerations

There is a known exploit for combination of a secure transport protocol and history-based compression [CRIME]. Implementors should give attention to this point when integrating this extension with other extensions or protocols.

10. IANA Considerations

10.1. Registration of the "permessage-deflate" WebSocket Extension Name

This section describes a WebSocket extension name registration in the WebSocket Extension Name Registry [RFC6455].

Extension Identifier
   permessage-deflate

Extension Common Name
   WebSocket Per-message Deflate

Extension Definition
   This document.

Known Incompatible Extensions
   None

The "permessage-deflate" extension name is used in the "Sec-WebSocket-Extensions" header in the WebSocket opening handshake to negotiate use of the permessage-deflate extension.

10.2. Registration of the "Per-message Compressed" WebSocket Framing Header Bit

This section describes a WebSocket framing header bit registration in the WebSocket Framing Header Bits Registry [RFC6455].

Header Bit
   RSV1

Common Name
   Per-message Compressed

Meaning
   The message is compressed or not.

Reference
   Section 6 of this document.

The "Per-message Compressed" framing header bit is used on the first fragment of non-control messages to indicate whether the payload of the message is compressed by the PMCE or not.

11. Acknowledgements

Special thanks to Patrick McManus who wrote up the initial specification of a DEFLATE-based compression extension for the WebSocket Protocol to which I referred to write this specification.

Thank you to the following people who participated in discussions on the HyBi WG and contributed ideas and/or provided detailed reviews (the list is likely to be incomplete): Adam Rice, Alexey Melnikov, Arman Djusupov, Bjoern Hoehrmann, Brian McKelvey, Dario Crivelli, Greg Wilkins, Inaki Baz Castillo, Jamie Lokier, Joakim Erdfelt, John A. Tamplin, Julian Reschke, Kenichi Ishibashi, Mark Nottingham, Peter Thorson, Roberto Peon, Simone Bordet, Tobias Oberstein and Yutaka Hirano. Note that people listed above didn't necessarily endorse the end result of this work.

12. References

12.1. Normative References

[RFC1951] Deutsch, P., "DEFLATE Compressed Data Format Specification version 1.3", RFC 1951, May 1996.

[RFC5234] Crocker, D. and P. Overell, "Augmented BNF for Syntax Specifications: ABNF", STD 68, RFC 5234, January 2008.

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

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

[LZ77] Ziv, J. and A. Lempel, "A Universal Algorithm for Sequential Data Compression", IEEE Transactions on Information Theory, Vol. 23, No. 3, pp. 337-343.

12.2. Informative References

[RFC1979] Woods, J., "PPP Deflate Protocol", RFC 1979, August 1996.

[Zlib] Gailly, J. and M. Adler, "Zlib", <http://zlib.net/>.

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

Author's Address

Takeshi Yoshino
Google, Inc.

Email: tyoshino@google.com