Network Working Group
Internet-Draft
Intended status: Informational
Expires: April 21, 2017

I. Svirid

October 18, 2016

WebSocket2 over HTTP/2
draft-svirid-websocket2-over-http2-00

Abstract

This document specifies a new protocol called WebSocket2 ontop of HTTP/2. The WebSocket2 protocol enables two-way binary communication between a client running untrusted code in a controlled environment to a remote host that has opted-in to communications from that code.

This protocol has little incommon with the WebSocket protocol [RFC6455] other than client side API compatibility.

Please send feedback to the ietf-http-wg@w3.org mailing list.

Status of This Memo

This Internet-Draft is submitted 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 April 21, 2017.

Copyright Notice

Copyright (c) 2016 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (http://trustee.ietf.org/license-info) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect

to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction

You can read about why two way data streaming is important from the introduction to WebSockets in RFC6455 Section 1 <https://tools.ietf.org/html/rfc6455#section-1>.

WebSocket2 over HTTP/2 is a protocol ontop HTTP/2 designed for modern times. Previous WebSocket client side API will be fully backward compatible with WebSocket2.

In this document, we describe WebSocket2 and how to layer WebSocket2 semantics onto HTTP/2 semantics by defining detailed mapping, replacement of operations and events.

1.1. Conventions 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 RFC 2119 [RFC2119].

2. Overview

WebSocket2 is functionally equivalent to binary streaming between a sandboxed client and inexplicit host, but layered on top of HTTP2.

Key advantages of WebSocket2 over WebSocket:

o No masking

o Simpler handshake and negotiation

o Lz4 compression method

Key advantages of WebSocket2 over HTTP/2 include:

o Two-way real time communication

o Server push and delivery without client request

o Binary transmission

2.1. WebSocket2 Protocol

The protocol has two main parts, the handshake and data transfer. Data transmitted using WebSocket2 supports compression.

3. Handshake

The main job of the handshake is to request authority and if granted by the server, to negotiate a compression medium.

3.1. Client Handshake Request

The client MUST use the :method GET.

The client MUST send a sec-ws2-version header that MUST specify the websocket2 version being used.

The client MAY send a sec-ws2-compression header that advertises the compression methods the client supports. Valid key value pairs include:

- o lz4=1-9;
  - * This client supports lz4 with compression levels from 1 to 9
- o lz4=1;
  - * This client supports lz4 with compression levels 1
- o deflate=8-15;
  - * This client supports deflate with sliding window bits from 8-15

Duplicate keys MUST NOT be present.

The client MUST NOT set the END_STREAM flag when sending the headers.

A client handshake request may look like:

```
:method: GET
:scheme: wss
:authority: example.org
:path: /demo
sec-ws2-version: 1
sec-ws2-compression: lz4=1-9; deflate=8-15;
```

3.2. Server Handshake Reply

END_STREAM on the HTTP/2 Header frame MUST only be set in the case of rejection.

The server MUST send ONLY ONE of the advertised compression methods or exclude the sec-ws2-compression header from the response, signaling that no compression will be used.

The server MUST include the sec-ws2-error header in the reply with an outlined error reason.

A successful server handshake reply may look like:

```
:status: 200
sec-ws2-compression: lz4=1;
sec-ws2-error: success
```

This signals that the server chose to use lz4 with a compression level of 1. Now both the client and server MUST use only this compression method.

3.2.1. Handshake Error Reasons

Valid error reasons are:

success
* The server accepted the client

invalid_version
* This version of websockets is not supported by the server

cannot_negotiate_compression
* This means the client did not offer any compression that the server requires

rejected
* This means the server rejected the client and does not want to say why. This means the server supports websockets, but rejected this particular client

3.3. Post Handshake

Following the handshake the client or server MUST NEVER set the END_STREAM flag on any HTTP/2 DATA frame UNLESS the stream is to be gracefully terminated. Only a HTTP/2 DATA frame containing a WebSocket2 error frame allow the END_STREAM flag to be set.

4. Data Framing

Once a handshake has been successfully completed the remote endpoints can begin to send data to each other. Data is sent using the HTTP/2 transport layer fully adhering to DATA Frames, Section 6.1 [RFC7540]. WebSocket2 has its own encapsulated framing protocol that is not to be confused with HTTP/2 DATA Frames.

Three frame types are defined:

text (represented by 0)
binary (represented by 1)
error (represented by 2)

Three compression types are defined:

none (represented by 0)
lz4 (represented by 1)
deflate (represented by 2)

A WebSocket2 over HTTP/2 frame starts with the full frame length in a special format called VarSize. If the first octet is less than 254,

this is the full frame length. If the first octet is 254, read the next 16 bits as a little endian unsigned number for the frame length. If the first octet is 255, read the next 32 bits as a little endian unsigned number for the frame length.

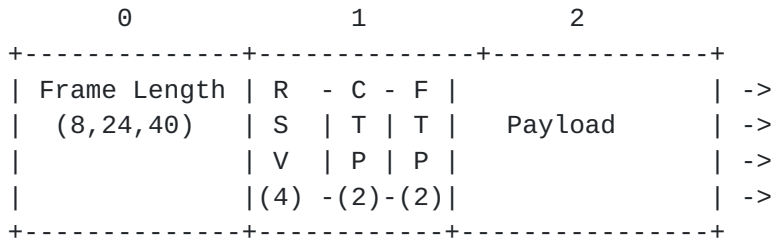
Next are 4 reserved bits that MUST be set to 0.

The next 2 bits specify the compression type.

The next 2 bits specify the frame type.

Next is the payload which MAY be compressed if compression was negotiated.

The term PAYLOAD in this section refers to data AFTER decompression if compression was negotiated.



4.1. Text Frame

The text frame has a Frame Type value of 0. The payload must be UTF-8 encoded text. The whole message MUST contain valid UTF-8. Invalid UTF-8 in the payload requires the remote end point to send an error frame and close their side of the stream.

4.2. Binary Frame

The binary frame has a Frame Type value of 1. The payload must be arbitrary binary data which the application layer passes without comprehension.

4.3. Error Frame

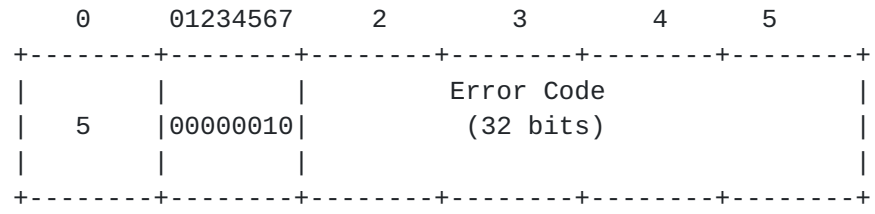
The error frame has a Frame Type value of 2. It MUST contain a VALID 32 bit error code. If the code is shorter than 32 bits, 0 bits MUST make up the rest to fill a total of 32 bits. Currently there are no error codes less than 32 bits.

The HTTP/2 transport layer DATA frame carrying the WebSocket2 error frame MUST have the END_STREAM flag set.

No further WebSocket2 frames may be sent from this point onward and the stream is half closed.

The remote endpoint that receives the error frame MUST flush all pending sends followed by an error frame of its own with the CLOS error code. The HTTP/2 Data frame carrying this WebSocket2 frame MUST have the END_STREAM flag set.

The full WebSocket2 error frame with the length:



4.3.1. Error Frame Codes

Valid error frame codes currently are:

CLOS

* This should be sent when a client or server want to close the stream gracefully.

UTF8

* This should be sent when invalid UTF8 was passed in a text frame by a remote endpoint.

DECO

* This should be sent when decompression failed by a remote endpoint.

FRAM

* This should be sent when an invalid Frame Type was specified.

LRGE

* This should be sent when an endpoint rejects a frame due to it being too large. This is up to the endpoint.

5. VarSize

VarSize is a variable sized binary ranging from 1-5 octets. It represents an unsigned value. If the first octet is equal to 254, read the next 16 bits as little endian unsigned. If the first octet is equal to 255, read the next 32 bits as little endian unsigned. Otherwise if the first octet is less than 254, treat this as the final value.

6. Compression

WebSocket defined one compression method which used deflate and kept a sliding window. This compression is great but has limitations. Also keeping a sliding window is memory intensive.

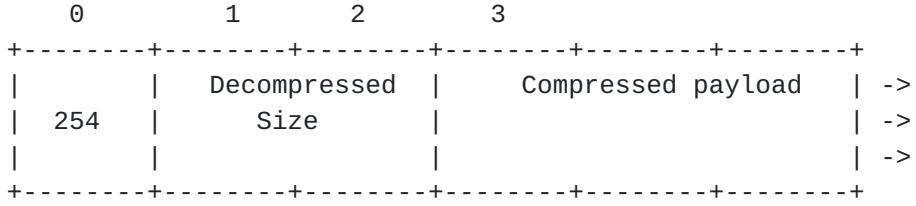
Lz4 is a compression method that is great for any kind of data while being very cheap.

Currently two compression methods are defined lz4 and deflate. The protocol is open to tweaking or accepting more in the future.

6.1. LZ4 Compressed Payload

A lz4 compressed payload is a VarSize number of the decompressed size followed by the actual compressed payload. The lz4 compression level to use MUST be what was negotiated in the handshake.

A lz4 compressed payload may look like:



6.2. Deflate Compressed Payload

The deflate compression method implements [RFC7692] but defines more strict bounds. There is no ability to reset compression context.

* Both client and server MUST use the sliding window bits as determined by the server.
* The client MUST use a memory level of 8.
* The client MUST use a compression level of 9 (best_compression).

Any sliding window MUST NEVER have its context reset.

If the deflated payload trailing 4 octets are 0x00, 0x00, 0xFF, 0xFF, remove them before sending the payload.

Before inflating the payload append 0x00, 0x00, 0xFF, 0xFF to the end of it.

7. References

7.1. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <http://www.rfc-editor.org/info/rfc2119>.

[RFC7692] Yoshino, T., "Compression Extensions for WebSocket", RFC 7692, DOI 10.17487/RFC7692, December 2015, <http://www.rfc-editor.org/info/rfc7692>.

7.2. Informative References

[RFC6455] Fette, I. and A. Melnikov, "The WebSocket Protocol", RFC 6455, DOI 10.17487/RFC6455, December 2011, <http://www.rfc-editor.org/info/rfc6455>.

[RFC7540] Belshe, M., Peon, R., and M. Thomson, Ed., "Hypertext Transfer Protocol Version 2 (HTTP/2)", RFC 7540, DOI 10.17487/RFC7540, May 2015, <http://www.rfc-editor.org/info/rfc7540>.

Appendix A. Acknowledgements

The author wishes to thank Kari hurtta for contributing the handshake.

The author wishes to thank the participants of the WebSocket protocol, participants of the HTTP/2 protocol and participants of the QUIC protocol.

Author's Address

Ivan Sviridov
Toronto, ON
Canada

Email: vans_163@yahoo.com