HTTPbis Working Group
Internet-Draft
Expires: November 23, 2014

M. Bishop
Microsoft

May 22, 2014

Extension Frames in HTTP/2
draft-bishop-http2-extension-frames-01

Abstract

This document describes a proposed modification to the HTTP/2 specification to better support the creation of extensions without the need to version the core protocol or invoke additional protocol identifiers.

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 November 23, 2014.

Table of Contents

1. Introduction

HTTP/2 previously offered an inconsistent story about the use of extensions. Following a discussion of the previous version of this draft, the working group reached consensus to prohibit all extensibility, declaring that any new functionality would constitute the creation of a new protocol with a new ALPN identifier. This was driven in large part by a desire not to delay the specification while an extension model was finalized and implemented.

In the wake of this decision, a number of new frames and subfeatures have been proposed (BLOCKED (Appendix A.1), ALTSVC (Appendix A.2), compression of DATA frames (Appendix A.3)), delaying the specification and introducing a dependency on a draft [I-D.ietf-httpbis-alt-svc] which is not as close to ready for last call as the core HTTP/2 specification. Others, such as DRAINING, have been suggested on the mailing list though not introduced to the specification. Many of these features are optional either to use or to process, limiting their broad applicability.

In the words of Antoine de Saint-Exupery, "Perfection is achieved, not when there is nothing more to add, but when there is nothing left to take away." Many of these frames represent concerns which, while worth addressing, are not fundamental to the goals of HTTP/2. The HTTP/2 specification should be the minimal set of features which enables two peers to communicate efficiently and achieve the goals laid out in the working group charter.

This working group is empowered by its charter to work on additional extensions to HTTP provided that "[t]he Working Group Chairs ...

believe that it will not interfere with the work described above [definition of HTTP/2]." The working group is explicitly prohibited from defining HTTP/2 extensions until the HTTP/2 work is complete.

This draft contends that some or all of these late-breaking features could be easily recast as extensions, simplifying and unblocking the core specification. Existing implementations of these features would test the extensibility model in the process of interoperating with others who have chosen not to implement them, permitting us to finalize HTTP/2 and turn our attention to the set of extensions the working group has already reached consensus should be explored.

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].

All numeric values are in network byte order. Values are unsigned unless otherwise indicated. Literal values are provided in decimal or hexadecimal as appropriate. Hexadecimal literals are prefixed with "0x" to distinguish them from decimal literals.

2. Problems an extension model must solve

Possible extensions vary along a number of pivots. Any extension model must define how extensions will work along each pivot, which may include disallowing certain combinations. Possible variants of extensions include:

- Changes session state, or strictly informative
- Flow-controlled third-party data or freely-sent control data
- Hop-by-hop or end-to-end

There is no way to know whether the peer supports a given extension before sending extension-specific information. This can be simply addressed by saying that implementations MUST ignore frame types and settings values they don't understand. However, this model only works for strictly informative frames and/or settings. An extension model must define a way to determine whether a peer supports a given extension, if non-informative extensions are supported.

Another concern is that with only 256 frame types, the frame type space may be exhausted if many extensions are defined. Different extensions could collide with each other in the choice of frame type identifiers, since the space is limited. RFC 6709 [RFC6709]

discusses in detail the trade-offs that must be considered by any protocol's extension model. One key risk is that a difficult registration process will encourage the use of unregistered extensions, which leads to collisions, but a small space requires rigorous control over the identifier space.

Another risk that arises when extension is overly constrained is the emergence of protocol variations. RFC 6709 [RFC6709] has this to say: "Protocol variations -- specifications that look very similar to the original but don't interoperate with each other or with the original -- are even more harmful to interoperability than extensions." Where no extensions are possible, implementers who wish to extend HTTP/2 will quickly move to define a new protocol which looks remarkably similar to HTTP/2, but is not interoperable.

Future protocols using the HTTP/2 framing layer will face exactly the same problem as extension authors, since they share a frame type and setting value space with any extensions. Thus, a new frame introduced with, for example, HTTP/3 must avoid collision with any HTTP/2 extensions and must deal with space exhaustion. Any means of resolving such adoption after the fact complicates forward-porting of existing extensions.

This document proposes an alternative method of supporting extension frames and settings, with the following goals:

- Reduce the probability of collision among extensions and between extensions and future versions of HTTP
- Enable peers to quickly discover support for a particular extension on the far side
- Enable extension implementers to interoperate with minimal procedural overhead

3. Extension Functionality

3.1. Extension Identification and Negotiation

An extension to HTTP/2 is identified by an Extension ID. An Extension ID is a 32-bit identifier registered with IANA. Extension identifiers above 0xFFFF0000 are reserved for experimental use, as described below (Section 3.1.1).

3.1.1. Experimental Extensions

The designer of an extension MAY self-allocate an extension ID in the experimental range defined above without interaction with IANA. Such an extension MUST use only frame numbers and setting IDs from the experimental range. These extensions MUST NOT be generally deployed until a non-experimental extension ID has been allocated.

As a further guard against accidental collisions, an experimental extension SHOULD define a random 32-bit number, and include this number as the first four bytes of each frame used by the extension. Received frames which do not include this identifier MUST be treated as an unrecognized frame type.

3.2. Extension Negotiation

An implementation which supports extensions SHOULD send an EXTENSIONS (Section 3.3.2) frame immediately following its SETTINGS frame at connection establishment, listing all extensions that it wishes to use during the lifetime of the connection. After receiving a corresponding EXTENSIONS frame, any extensions which were present in both frames are considered to be in effect for the lifetime of the connection. The EXTENSIONS frame may be sent only once per connection.

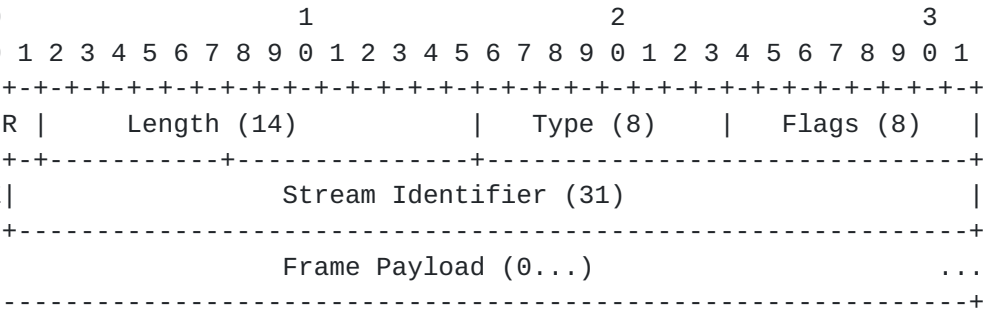
An empty EXTENSIONS frame declares that the sender does not wish to employ any hop-by-hop extensions beyond the negotiated protocol.

Extension-defined hop-by-hop frames and settings which modify stream or session state (including flow control) MUST NOT be sent until the EXTENSIONS frame has been received from the remote endpoint declaring support for the associated extension ID. Extension-defined frames and settings which are strictly informative MAY be sent between sending the EXTENSIONS frame and before receiving the peer's EXTENSIONS frame. Implementations SHOULD NOT send informative frames or settings from any extension after receiving an EXTENSIONS frame which does not list support for that extension, since the receiver likely will not understand the extra information.

3.3. New Frames and Modifications

3.3.1. Definition of Frames

To support the notion of end-to-end extension frames, one Reserved bit from the Frame Header is given a defined meaning:



Frame Header

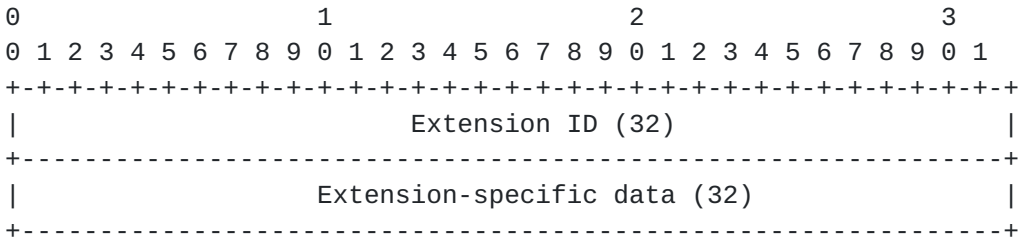
The newly-added "E" field marks whether a frame is intended for end-to-end transmission or hop-by-hop transmission. End-to-end frames MUST be relayed by intermediaries, even if the frame type is unknown. Such frames do not imply any changes to stream or session state. End-to-end frames are always subject to flow control.

An end-to-end frame on stream zero is meaningless, and MUST be discarded upon receipt.

Of the frames defined in the base HTTP/2 spec, DATA frames MUST set the E bit; all other control frames (WINDOW_UPDATE, PUSH_PROMISE, HEADERS, etc.) MUST NOT set the E bit. Receipt of a base HTTP/2 frame with the E bit set improperly indicates a fundamental error in the remote implementation, and MUST trigger a connection error of type PROTOCOL_ERROR.

3.3.2. EXTENSIONS Frame

The EXTENSIONS frame (number TBD) carries a list of zero or more extensions supported by the sender:

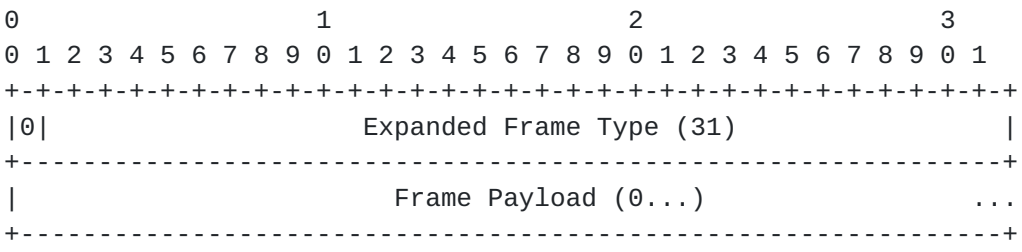


EXTENSIONS format

For each extension, the sender includes 32 bits of inital state. The semantics of this value are completely defined by the extension.

3.3.3. EXPANDED Frame

The EXPANDED frame (number 0xFF) expands the space of frame types by supplying additional bits for the frame type:



EXPANDED Format

In order to mitigate the concern that 256 frame types are too few to allow free access to extensions, the EXPANDED frame defines an additional 31 bits that can be used for the frame type space. Frame types numbered 256 or greater are encoded within an EXPANDED frame, and the Expanded Frame Type field set to the desired frame type value minus 256. This increases the maximum frame type to 0x80000100 without increasing the frame size for common frame types.

Implementations which have no knowledge of frame types greater than 255 MAY ignore any EXPANDED frames upon receipt, though intermediaries MUST still relay end-to-end EXPANDED frames.

3.3.4. Extension-Defined Frames

An extension MAY define new frame types, which are registered with IANA. Frame types greater than 0x80000000 will not be allocated by IANA and are reserved for use by experimental extensions. Frame types less than 256 are reserved for assignment by standards-track RFCs.

As part of the definition of the extension and frame type, the extension MUST specify whether the frames it defines modify session state in any way, including being flow-controlled. (Any frame which modifies session state MUST NOT be sent prior to receipt of an EXTENSIONS frame declaring support for the specified extension.)

Only frames which do not change stream or session state may be marked as end-to-end, since intermediaries which do not understand the frame

type would not be able to track the state changes. Because end-to-end frames have unknown payload and provenance, end-to-end frames are always flow-controlled.

Frames which do not modify stream or session state MAY be sent at any time. However, an implementation SHOULD NOT send hop-by-hop extension frames after receiving an EXTENSIONS frame indicating that the other party will not understand the frame being sent.

An extension has complete freedom to define the payload, flags, and other semantics of the frames it specifies, including when and on what streams the frame may or may not be sent.

3.3.4.1. Handling by Intermediaries

Intermediaries MUST forward all end-to-end frames regardless of whether they recognize the frame type. Endpoints (user agents and origin servers) MUST discard any frame types which they do not recognize. Such frames are, by definition, informational and can be safely ignored without affecting the shared state with the sender.

All hop-by-hop extension-defined frames MUST be dropped by intermediaries which do not support the extension. However, each extension SHOULD specify how an intermediary translates the frames defined by the extension toward other peers which might or might not support the same extension. When an intermediary advertises support for an extension, it MUST abide by the extension-defined intermediary behavior.

An intermediary which advertises support for an extension is explicitly not guaranteeing that all peers to which it will relay information support the same extensions. Extension definitions SHOULD define how intermediaries translate in the following situations:

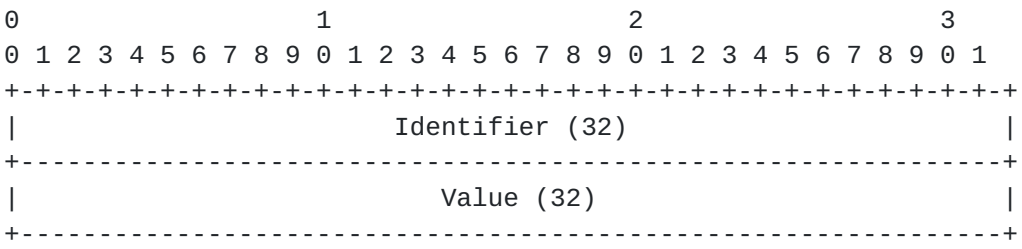
Relaying to HTTP/1.1 connection

Relaying to HTTP/2 connection without extension support

Relaying to HTTP/2 connection with extension support

3.4. Settings

This draft restores the definition of a setting value as follows:



Modified Setting Value Format

Extensions may define settings whose identifiers are registered with IANA. The semantics of any such setting value are defined strictly by the extension. Implementations MUST ignore unknown settings and MUST NOT emit settings defined by an extension which has not been announced in an EXTENSIONS (Section 3.3.2) frame.

4. IANA Considerations

This draft proposes the restoration to the HTTP/2 spec of IANA registries for the following, pre-populated with the values defined in the HTTP/2 specification:

- Frame types, with values less than 256 restricted to standards-track RFCs and values greater than 0x80000000 reserved for private experimental use

- Setting identifiers, with values greater than 0xFFFF0000 reserved for private experimental use

- Error codes

And additionally, the creation of a registry for Extension IDs, with values above 0xFFFF0000 reserved for private experimental use.

Given the expanded space, these registries should be allocated on a first-come-first-served [RFC5226] basis except as described above, though a publicly-available specification for each extension is strongly recommended.

5. References

[I-D.ietf-httpbis-alt-svc]
Nottingham, M., McManus, P., and J. Reschke, "HTTP Alternative Services", draft-ietf-httpbis-alt-svc-01 (work in progress), April 2014.

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

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

[RFC6709] Carpenter, B., Aboba, B., and S. Cheshire, "Design Considerations for Protocol Extensions", RFC 6709, September 2012.

Appendix A. Example Extensions

This section describes how certain extensions might leverage the above model. Because a number of recent additions to the HTTP/2 specifications are excellent candidates for extension definition, they are used as examples here.

A.1. Blocked Flow-Control Announcement Extension

A.1.1. EXTENSIONS Payload

Support for the Blocked Flow Control Announcement Extension is indicated by including extension ID 0xB39D237F in an EXTENSIONS frame. The initial data in the EXTENSIONS frame MUST be zero when sent and MUST be ignored on receipt.

A.1.2. BLOCKED Frame

The BLOCKED frame defines no flags and contains no interpretable payload. Because the frame is experimental, each BLOCKED frame MUST contain the static payload 0xABED6142 for disambiguation. A receiver MUST treat the receipt of a BLOCKED frame with any other payload as an unknown frame type and ignore it.

A.1.3. Use of BLOCKED Frame

The BLOCKED frame is used to provide feedback about the performance of flow control for the purposes of performance tuning and debugging. The BLOCKED frame can be sent by a peer when flow controlled data cannot be sent due to the connection- or stream-level flow control window being zero or less. This frame MUST NOT be sent if there are other reasons preventing data from being sent, such as a lack of available data or the underlying transport being blocked.

The BLOCKED frame MAY be sent on a connection prior to receiving an EXTENSIONS frame, but SHOULD NOT be sent after the receipt of an EXTENSIONS frame which does not include the BLOCKED extension ID.

The BLOCKED frame is sent on the stream that is blocked, that is, the stream with a non-positive number of bytes available in the flow control window. A BLOCKED frame can be sent on stream 0x0 to indicate that connection-level flow control is blocked.

An endpoint MUST NOT send any subsequent BLOCKED frames until the affected flow control window becomes positive. This means that WINDOW_UPDATE frames are received or SETTINGS_INITIAL_WINDOW_SIZE is increased before more BLOCKED frames can be sent.

A.1.4. Behavior by Intermediaries

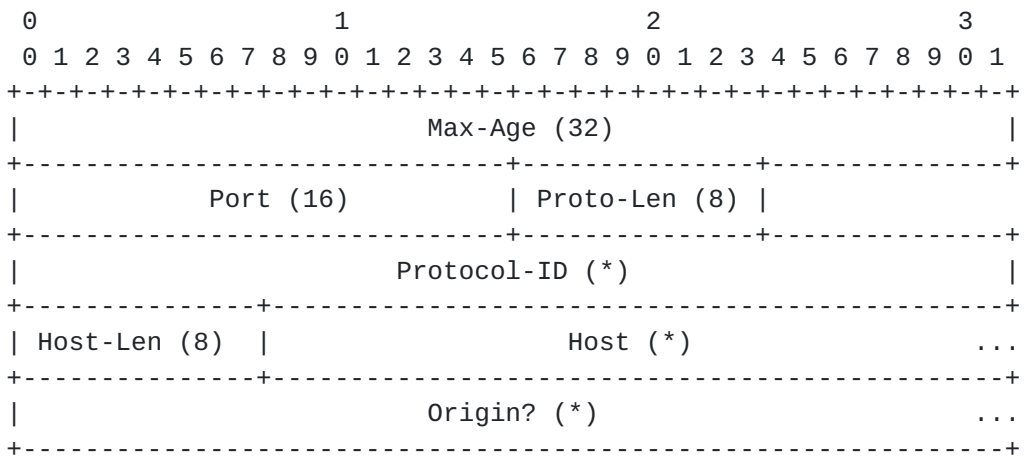
Because flow-control is hop-by-hop, intermediaries MUST NOT relay a BLOCKED frame onto any other connection. If the intermediary is blocked by flow control, they MAY generate BLOCKED frames independently on other connections where BLOCKED is supported.

A.2. Alternate-Service Announcement

A.2.1. EXTENSIONS Payload

Support for the Alternate Service Announcement Extension is indicated by including extension ID 0x8877B974 in an EXTENSIONS frame. The initial data in the EXTENSIONS frame MUST be zero when sent and MUST be ignored on receipt.

A.2.2. ALTSVC Frame



ALTSVC Frame Payload

The ALTSVC frame contains the following fields:

Max-Age: An unsigned, 32-bit integer indicating the freshness lifetime of the alternative service association.

Port: An unsigned, 16-bit integer indicating the port that the alternative service is available upon.

Proto-Len: An unsigned, 8-bit integer indicating the length, in octets, of the Protocol-ID field.

Protocol-ID: A sequence of bytes (length determined by "Proto-Len") containing the ALPN protocol identifier of the alternative service.

Host-Len: An unsigned, 8-bit integer indicating the length, in octets, of the Host field.

Host: A sequence of characters (length determined by "Host-Len") containing an ASCII string indicating the host that the alternative service is available upon. An internationalized domain name MUST be expressed using A-labels.

Origin: An optional sequence of characters (length determined by subtracting the length of all preceding fields from the frame length) containing the ASCII serialisation of an origin that the alternate service is applicable to.

The ALTSVC frame does not define any flags.

A.2.3. Use of ALTSVC Frame

The ALTSVC frame (type=0xA) advertises the availability of an alternative service to the client. It can be sent at any time for an existing client-initiated stream or stream 0, and is intended to allow servers to load balance or otherwise segment traffic; see [I-D.ietf-httpbis-alt-svc] for details.

An ALTSVC frame on a client-initiated stream indicates that the conveyed alternative service is associated with the origin of that stream.

An ALTSVC frame on stream 0 indicates that the conveyed alternative service is associated with the origin contained in the Origin field of the frame. An association with an origin that the client does not consider authoritative for the current connection MUST be ignored.

The ALTSVC frame is intended for receipt by clients; a server that receives an ALTSVC frame MAY treat it as a connection error of type PROTOCOL_ERROR.

A server MAY send an ALTSVC frame before receiving an EXTENSIONS frame listing support for the Alternate-Service Availability Announcement extension, but SHOULD NOT send an ALTSVC frame after receiving an EXTENSIONS frame which does not declare support.

A.2.4. Behavior by Intermediaries

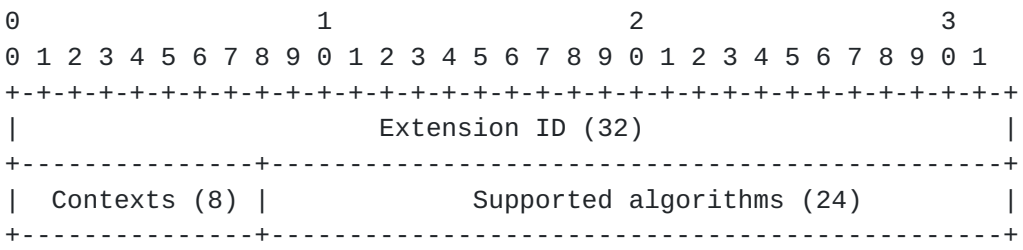
The ALTSVC frame is processed hop-by-hop. An intermediary MUST NOT forward ALTSVC frames, though it can use the information contained in ALTSVC frames in forming new ALTSVC frames to send to its own clients.

A.3. Compressed Data Frames

The COMPRESSED_DATA frame (type=TBD) permits a frame-by-frame choice of transfer encoding, permitting connections to employ compression where appropriate while still enabling the separation of different data into different contexts as appropriate.

A.3.1. EXTENSIONS Payload

Support for the Compressed Data Extension is indicated by the following in an EXTENSIONS frame:



Compressed Data in EXTENSIONS

The EXTENSIONS entry contains the following fields:

Extension ID: The extension ID for the Compressed Data Extension is 0x53F9F537.

Contexts An eight-bit count of the number of separate compression contexts the sender is willing to maintain. This SHOULD be greater than zero.

Supported algorithms A bitmap of compression algorithms supported by the sender:

0x001: Compress The "compress" coding is an adaptive Lempel-Ziv-Welch (LZW) coding that is commonly produced by the UNIX file compression program "compress".

0x002: Deflate The "deflate" coding is a "zlib" data format [RFC1950] containing a "deflate" compressed data stream [RFC1951] that uses a combination of the Lempel-Ziv (LZ77) compression algorithm and Huffman coding.

0x003: GZip The "gzip" coding is an LZ77 coding with a 32 bit CRC that is commonly produced by the gzip file compression program [RFC1952].

Other bits: Reserved for future updates; MUST be zero when sent and ignored upon receipt

Setting the corresponding bit indicates that the sender supports creating a compression context for the corresponding algorithm.

A.3.2. COMPRESSED_DATA Frame

The COMPRESSED_DATA frame defines the following flags:

END_STREAM (0x1): Bit 1 being set indicates that this frame is the last that the endpoint will send for the identified stream. Setting this flag causes the stream to enter one of the "half closed" states or the "closed" state.

END_SEGMENT (0x2): Bit 2 being set indicates that this frame is the last for the current segment. Intermediaries MUST NOT coalesce frames across a segment boundary and MUST preserve segment boundaries when forwarding frames.

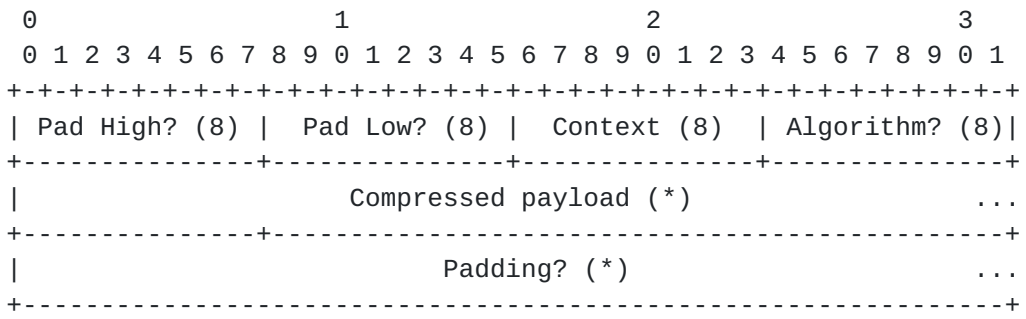
PAD_LOW (0x8): Bit 4 being set indicates that the Pad Low field is present.

PAD_HIGH (0x10): Bit 5 being set indicates that the Pad High field is present. This bit MUST NOT be set unless the PAD_LOW flag is also set. Endpoints that receive a frame with PAD_HIGH set and PAD_LOW cleared MUST treat this as a connection error of type PROTOCOL_ERROR.

Init_Context (0x20): Indicates the presence of the Algorithm and Initialization fields in the COMPRESSED_DATA frame. MUST be set on the first frame to reference a context which has not previously been used, or which has been cleared. If set on any other frame, the previous value of the context MUST be discarded before further processing.

Clear_Context (0x40): Indicates that this is the last frame which will use the current context state, and that the context MUST be discarded after interpretation of the current frame.

The payload is formatted as follows:



COMPRESSED_DATA

The fields are:

Pad High: An 8-bit field containing an amount of padding in units of 256 octets. This field is optional and is only present if the PAD_HIGH flag is set. This field, in combination with Pad Low, determines how much padding there is on a frame.

Pad Low: An 8-bit field containing an amount of padding in units of single octets. This field is optional and is only present if the PAD_LOW flag is set. This field, in combination with Pad High, determines how much padding there is on a frame.

Context: An eight-bit value reflecting which of the connection's contexts the sender intends to use. This MUST be less than the value the recipient declared in EXTENSIONS.

Algorithm: Present only if the Init_Context flag is set. Declares the compression algorithm the sender intends to employ on the new context. Integer taken from the same list employed in the EXTENSIONS frame to announce support.

Compressed payload The remainder of the payload is the compressed content, interpreted using the selected compression context.

Padding: Padding octets that contain no application semantic value. Padding octets MUST be set to zero when sending and ignored when receiving.

A.3.3. Use of COMPRESSED_DATA Frame

The COMPRESSED_DATA frame may be sent instead of a DATA frame provided that both of the following are true:

- The sender is allowed to send a DATA frame at this time on this stream
- The recipient has advertised support for the Compressed Data Extension

The sender SHOULD clear contexts, use different contexts, or compress selectively to prevent attacker-controlled data from being compressed in the same compression context as server-controlled data.

COMPRESSED_DATA frames are subject to flow control and can only be sent when a stream is in the "open" or "half closed (remote)" states. The entire frame payload is included in flow control, including Pad Low, Pad High, Context, Algorithm, Initialization, and Padding fields if present. If a COMPRESSED_DATA frame is received whose stream is not in "open" or "half closed (local)" state, the recipient MUST respond with a stream error of type STREAM_CLOSED. After processing flow control, the frame is decompressed and the result processed as if a DATA frame with the decompressed payload had just been received.

If the COMPRESSED_DATA frame had the Clear_Context flag set, the sender MUST discard the compression context immediately following compression, and the recipient MUST do likewise immediately after decompression.

A.3.4. Behavior by Intermediaries

COMPRESSED_DATA frames are processed hop-by-hop, though an intermediary MAY relay the same compressed content onto another connection if an identical compression context is available.

Intermediaries MAY convert COMPRESSED_DATA frames to use different compression schemes on different connections, and MAY convert COMPRESSED_DATA frames into DATA frames on connections which do not support this extension or which do not support a compression algorithm to which the intermediary is willing to convert.

Intermediaries MUST NOT convert DATA frames into COMPRESSED_DATA frames.

Appendix B. Acknowledgements

This document includes input from Rob Trace, Gabriel Montenegro, and James Snell.

Sample extensions are based largely on the work of Mark Nottingham, Roberto Peon, and Matthew Kerwin.

Author's Address

Mike Bishop
Microsoft

EMail: michbish@microsoft.com