QUIC Working Group
Internet-Draft
Intended status: Standards Track
Expires: June 17, 2018

M. Bishop
Akamai

December 14, 2017

Header Compression for HTTP/QUIC
draft-bishop-quic-http-and-qpack-07

Abstract

HTTP/2 [RFC7540] uses HPACK [RFC7541] for header compression. However, HPACK relies on the in-order message-based semantics of the HTTP/2 framing layer in order to function. Messages can only be successfully decoded if processed by the decoder in the same order as generated by the encoder. This draft refines HPACK to loosen the ordering requirements for use over QUIC [I-D.ietf-quic-transport].

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 https://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 June 17, 2018.

Table of Contents

1. Introduction

HPACK has a number of features that were intended to provide performance advantages to HTTP/2, but which don’t live well in an out-of-order environment such as that provided by QUIC.

The largest challenge is the fact that elements are referenced by a very fluid index. Not only is the index implicit when an item is added to the header table, the index will change without notice as other items are added to the header table. Static entries occupy the first 61 values, followed by dynamic entries. A newly-added dynamic entry would cause older dynamic entries to be evicted, and the retained items are then renumbered beginning with 62. This means that, without processing all preceding header sets, no index into the dynamic table can be interpreted, and the index of a given entry cannot be predicted.

Any solution to the above will almost certainly fall afoul of the memory constraints the decompressor imposes. The automatic eviction of entries is done based on the compressor's declared dynamic table size, which MUST be less than the maximum permitted by the decompressor (and relayed using an HTTP/2 SETTINGS value).

Further, streams in QUIC are lossy in the presence of stream resets. While HTTP/2 (via TCP) guarantees the delivery of all previously-sent data on a stream even if that stream is reset, QUIC does not retransmit lost frames if a stream has been reset, and may discard data which has not yet been delivered to the application.

Early versions of QPACK were small deltas of HPACK to introduce order-resiliency. Recent versions depart from HPACK more substantially to add resilience against reset message streams and reduce the impact of head-of-line blocking.

In the following sections, this document proposes a successor to HPACK which makes different trade-offs, enabling partial out-of-order interpretation and bounded memory consumption with minimal head-of-line blocking. None of the proposed improvements to HPACK (strongly-typed fields, binary compression of common header syntax) are currently included, but certainly could be.

1.1. Terminology

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

2. QPACK - Concepts

HPACK combines header table modification and message header emission in a single sequence of coded bytes. QPACK bifurcates these into three channels:

- Connection-wide sets of table update instructions sent on non-request streams
- Connection-wide feedback on stream and checkpoint state on a single non-request stream
- Non-modifying instructions which use the current header table state to encode message headers on request streams

Because the per-message instructions introduce no changes to the header table state, no state is lost if these instructions are discarded due to a stream reset. Because the updates to the header table supply their own order controls (the checkpoint logic), they can be processed in any order and therefore delivered as messages using unidirectional QUIC streams.

2.1. Changes to Static and Dynamic Tables

QPACK uses two tables for associating header fields to indexes. The static table is unchanged from [RFC7541]. Unlike in [RFC7541], the tables are not concatenated, but are referenced separately.

The dynamic table is a map from index to header field. Indices are arbitrary numbers between 1 and 2^27. Each insert instruction will specify the index being modified. While any index MAY be chosen for a new entry, smaller numbers will yield better compression performance.

With decoder consent (see Section 4.1), it is possible for QPACK instructions to arrive which reference indices which have not yet been defined. Such instructions MUST wait until the index definition has arrived. In order to guard against malicious peers, implementations supporting blocking SHOULD impose a time limit and treat expiration of the timer as a decoding error.

2.1.1. Dynamic Table State Synchronization

In order to ensure table consistency, all modifications of the header table occur as separate messages rather than on request streams. Request streams contain only indexed and literal header entries.

No entries are automatically evicted from the dynamic table. Size management is purely the responsibility of the encoder, which MUST NOT exceed the declared memory size of the decoder.

To simplify state management in the dynamic table, _checkpoints_ are introduced. A checkpoint is used to track entries added to the dynamic table and streams that reference those entries, rather than

maintaining the full state of which streams reference which table entries.

Checkpoints are unordered and have an identifier which MUST be unique among checkpoints which have not been dropped. Each checkpoint has a unidirectional stream which begins with its identifier and contains a series of updates associated with that checkpoint. These updates SHOULD be processed as they arrive; it is not necessary (and might not be desirable) to wait for all instructions associated with a checkpoint to arrive before beginning to process it.

The feedback stream is used to relay state transitions to the peer. For example, when a decoder is done processing a header block, it signals this using the HEADERS_DONE message. The encoder uses this information to track which checkpoints can be dropped.

2.1.1.1. Checkpoint Lifecycle

A checkpoint is created by opening a new checkpoint stream. This places the checkpoint in the NEW state for both encoder and decoder. The encoder typically has at least one checkpoint in the NEW state.

Flushing a checkpoint is a two-step operation. First, the checkpoint stream is closed. At that time, the encoder’s NEW checkpoint becomes PENDING. The decoder moves its NEW checkpoint directly to LIVE and responds with an ACK_FLUSH message on the feedback stream. When the encoder receives this message, its PENDING checkpoint becomes LIVE.

Unused entries are evicted indirectly, by dropping checkpoints. Before a checkpoint can be dropped, its state is changed to DYING. Changing a checkpoint’s state to DYING allows the checkpoint to age out. This is a strictly internal state on the encoder, and not visible to the decoder. A DYING checkpoint can be returned to LIVE at the encoder’s discretion if necessary.

The encoder can change a DYING checkpoint to DEAD (sending a DROP instruction) when it is no longer referenced by any outstanding header blocks. The encoder sends the DROP command to the decoder when it declares a checkpoint DEAD.

To ensure consistency, the decoder drops the corresponding checkpoint and responds with an ACK_DROP message only when it has fully received all instructions the encoder has issued up to that point. The encoder drops the DEAD checkpoint upon receipt of the ACK_DROP message.

When a checkpoint is dropped by encoder or decoder, the table entries it references are checked: if an entry is no longer referenced by any checkpoint, the entry is evicted.

Dropping a checkpoint and the entries associated with it is not limited to just the oldest checkpoint; any DYING checkpoint - as long as state transition rules are followed - may be dropped. This flexibility permits the encoder to use a number of strategies for entry eviction.

As long as the maximum dynamic table size is observed, new checkpoints can be created; no upper limit on the number of checkpoints is specified. A well-balanced spread of checkpoints permits the encoder to recycle entries effectively.

2.2. Encoding Constraints

2.2.1. Permitted References

When encoding headers on a request stream, an encoder MAY reference any static table entry or any dynamic header table entry referenced by a LIVE checkpoint. References to entries in NEW or PENDING checkpoints are permitted only if the client has set "SETTING_QPACK_BLOCKING_PERMITTED" (see Section 4.1).

If a decoder receives a reference to an empty slot in the dynamic table but has not sent "SETTING_QPACK_BLOCKING_PERMITTED", this MUST be treated as a stream error of type "ERROR_QPACK_INVALID_REFERENCE" if on a request stream. References to empty slots in the dynamic table on a checkpoint stream MUST be treated as a connection error of type "ERROR_QPACK_INVALID_REFERENCE".

References to DYING checkpoints are possible by returning the checkpoint to LIVE, but this is usually inadvisable. Table entries contained only in a DEAD checkpoint can never be referenced.

2.2.2. Header Table Size

As in HPACK, the dynamic table is constrained to the maximum size specified by the decoder. An attempt to add a header to the dynamic table or to create a new checkpoint which causes it to exceed the maximum size MUST be treated as an error by a decoder. To enable encoders to reclaim space, encoders can drop old checkpoints (see Section 2.1.1).

The total table size is calculated as follows:

- The size of each entry is calculated as in HPACK

- Each checkpoint that has not been removed, regardless of state, consumes 64 bytes

2.2.2.1. Table Size Changes

HTTP/QUIC prohibits mid-stream changes of settings. As a result, only one table size change is possible: From the value a client assumes during the 0-RTT flight to the actual value included in the server's SETTINGS frame. The assumed value is required to be either a server's previous value or zero. A server whose configuration has recently changed MAY overlook inadvertent violations of its maximum table size during the first round-trip.

In the case that the value has increased, either from zero to a non-zero value or from the cached value to a higher value, no action is required by the client. The encoder can simply begin using the additional space. In the case that the value has decreased, the encoder MUST move checkpoints to the DYING state which, upon removal, would bring the table within the required size.

Regardless of changes to header table size, the encoder MUST NOT create new checkpoints or add entries to the table which would result in a size greater than the maximum permitted. This can imply that no additions are permitted while waiting for old checkpoints to complete.

3. Wire Format

QPACK instructions occur on three stream types, each of which uses a separate instruction space.

The feedback stream is a bidirectional server-initiated stream used for acknowledgement of actions and checkpoint state management. Checkpoint streams are unidirectional streams from encoder to decoder. Both types of streams consist of a series of QPACK instructions with no message boundaries, preceded by a stream header for checkpoint streams.

Finally, the contents of HEADERS and PUSH_PROMISE frames on request streams reference the QPACK table state.

This section describes the instructions which are possible on each stream type.

3.1. Feedback Stream

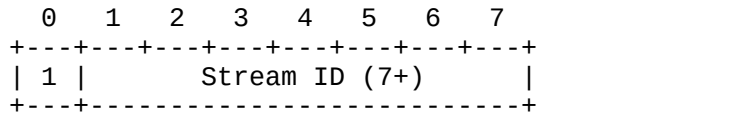
Stream 1, the first server-initiated bidirectional stream, is used as the feedback stream, since the client does not need to begin sending data on this stream until it has received data from the server.

This stream is critical to the HTTP/QUIC connection, and carries a stream of the instructions defined in this section. Data on this stream SHOULD be processed as soon as it arrives.

3.1.1. HEADERS_DONE

When the decoder has processed a frame containing header emission instructions (Section 3.3, HEADERS or PUSH_PROMISE frames) on a stream, it MUST emit a HEADERS_DONE message on the feedback stream. The same Stream ID can be identified multiple times, as multiple header-containing blocks can be sent on a single stream in the case of intermediate responses, trailers, pushed requests, etc.

Since header frames on a request stream are received and processed in order, this gives the encoder precise feedback on which header blocks within a stream have been fully processed. This information can then be used to correctly track outstanding stream references to checkpoints.

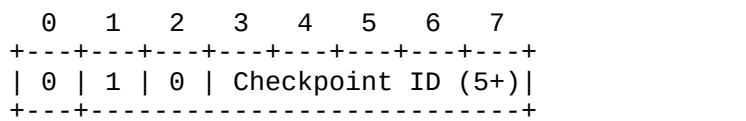


HEADERS_DONE instruction

3.1.2. ACK_FLUSH

When the decoder has finished processing all instructions that make up a checkpoint, it MUST indicate successful processing to the encoder by emitting an ACK_FLUSH instruction on the feedback stream.

Upon emitting an ACK_FLUSH, the checkpoint transitions from NEW to LIVE on the decoder. Upon receipt of an ACK_FLUSH, the checkpoint transitions from PENDING to LIVE on the encoder.

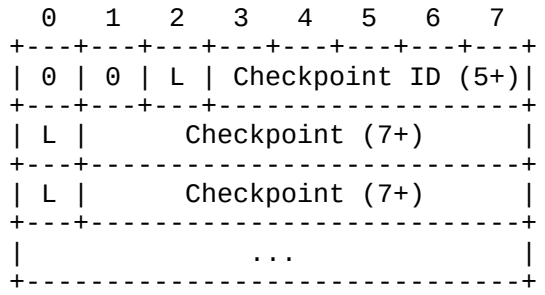


ACK_FLUSH instruction

3.1.3. DROP

When an encoder has received sufficient HEADERS_DONE messages to know that a DYING checkpoint has no outstanding references, it emits a DROP instruction to inform the decoder that the checkpoint can be removed. Upon sending a DROP instruction, a DYING checkpoint becomes DEAD. The DROP instruction also includes the IDs of any PENDING or NEW checkpoints which reference entries contained in the checkpoint being dropped. The "L" bit in each byte indicates whether another checkpoint ID follows (L=0) or this is the final byte of the DROP instruction (L=1).

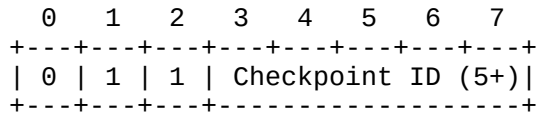
Upon receiving a DROP instruction, if all listed checkpoints have been fully processed (transitioned from NEW to LIVE), the identified LIVE checkpoint is immediately removed from the decoder state and an ACK_DROP instruction is emitted. Otherwise, the decoder saves the DROP instruction until other checkpoints become LIVE.



DROP instruction

3.1.4. ACK_DROP

When a decoder receives a DROP instruction, it removes the referenced checkpoint from its state and clears any table entries which were referenced only by that checkpoint. It then emits an ACK_DROP instruction. When an encoder receives an ACK_DROP instruction, it removes the corresponding DEAD checkpoint from its state and clears any table entries which were referenced only by that checkpoint.



ACK_DROP instruction

3.2. Checkpoint Streams

Each checkpoint stream indicates the creation and content of a NEW checkpoint. Each checkpoint has an ID; these IDs are chosen arbitrarily by the encoder, though lower values SHOULD be preferred. IDs of checkpoints which have been dropped MAY be reused for future NEW checkpoints.

When the encoder has finished writing all data on the stream, it changes the checkpoint to PENDING. When the decoder has received and processed all data on the stream, it changes the checkpoint to LIVE and generates an ACK_FLUSH.

Unidirectional streams in HTTP/QUIC begin with a stream header indicating the nature of the stream content; the identifier for QPACK checkpoints is 0x4B.

> *Note to readers:* This header does not currently exist in the main draft, but has manifested in several PRs, and would need to be resurrected.

Following the stream header, a checkpoint stream contains its checkpoint ID as an 8-bit prefix integer. The remainder of the stream’s data consists of the instructions defined in this section.

Data on checkpoint streams SHOULD be processed as soon as it arrives. If multiple checkpoint streams are received at once, a decoder SHOULD process data on each as it arrives if it has sent "SETTINGS_QPACK_BLOCKING_PERMITTED", but MAY process checkpoint streams one at a time.

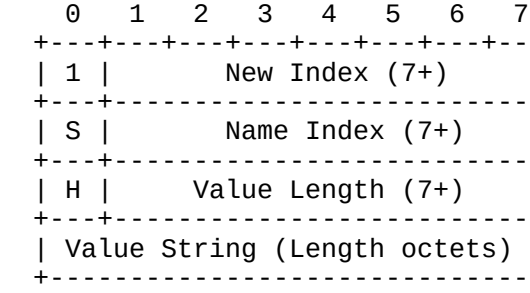
3.2.1. INSERT

An addition to the dynamic table starts with the ’1’ one-bit pattern, followed by the new index of the header represented as an integer with a 7-bit prefix. The decoder adds the supplied header to the checkpoint currently being processed, which is in the NEW state.

If the header field name matches the header field name of an entry stored in the static table or the dynamic table, the header field name can be represented using the index of that entry. In this case, the "S" bit indicates whether the reference is to the static (S=1) or dynamic (S=0) table and the index of the entry is represented as an integer with an 7-bit prefix (see Section 5.1 of [RFC7541]). This value is always non-zero.

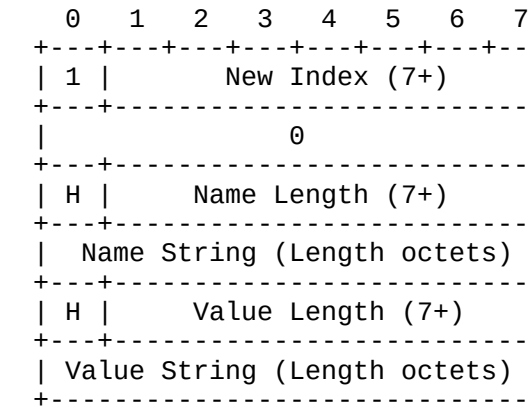
If an INSERT instruction uses an existing dynamic table entry for the name of an entry being added to the NEW checkpoint, both the existing

entry and the new entry are referenced by the NEW checkpoint. INSERT instructions which reference the dynamic table MUST reference only entries which are already included in a LIVE checkpoint. This avoids the possibility of one checkpoint stream blocking on a different checkpoint.



INSERT instruction -- Indexed Name

Otherwise, the header field name is represented as a string literal (see Section 5.2 of [RFC7541]). A value 0 is used in place of the table reference, followed by the header field name.



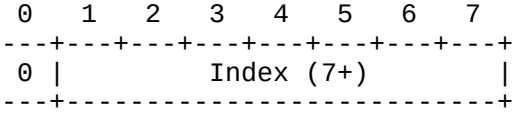
INSERT instruction -- New Name

Either form of header field name representation is followed by the header field value represented as a string literal (see Section 5.2 of [RFC7541]).

An encoder MUST NOT attempt to place a value at an index not known to be vacant. A decoder MUST treat the attempt to insert into an occupied slot or reference a name in a vacant slot as a fatal error.

3.2.2. TOUCH

This instruction is emitted to link a NEW checkpoint to an existing header table entry created by a previous checkpoint. This causes the entry not to be removed from the table so long as the current checkpoint is alive.



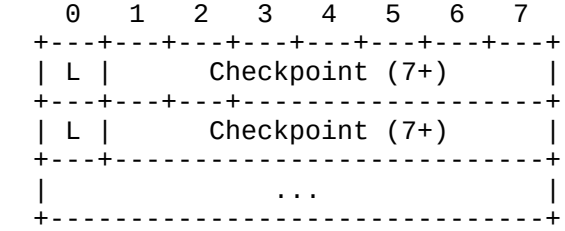
Indexed Header Field

The encoder SHOULD NOT issue multiple TOUCH commands for the same entry in the context of the same NEW checkpoint. If a non-existent index is specified, the decoder MUST treat is as an error.

3.3. Request Streams

Frames which carry HTTP message headers begin with an optional preface indicating potentially-blocking references in the frame. If present, this preface indicates that the request depends on one or more checkpoints which were NEW or PENDING for the encoder when the frame was generated. If these checkpoints are not LIVE on the decoder, it MAY delay reading the remainder of the frame until they are. (If any of these checkpoints have already been dropped, this must be treated as a stream error of type ERROR_QPACK_INVALID_REFERENCE.)

The preface is formatted as follows:

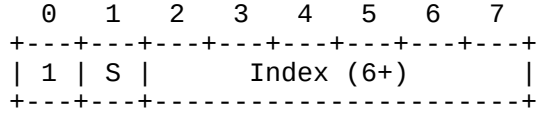


QPACK preface

The "L" bit indicates that this checkpoint is the last checkpoint in the preface; if the bit is unset (0), then another checkpoint follows.

3.3.1. Indexed Header Field Representation

An indexed header field representation identifies an entry in either the static table or the dynamic table and causes that header field to be added to the decoded header list, as described in Section 3.2 of [RFC7541].



Indexed Header Field

An indexed header field starts with the ’1’ 1-bit pattern, followed by the "S" bit indicating whether the reference is into the static (S=1) or dynamic (S=0) table. Finally, the index of the matching header field is represented as an integer with a 6-bit prefix (see Section 5.1 of [RFC7541]).

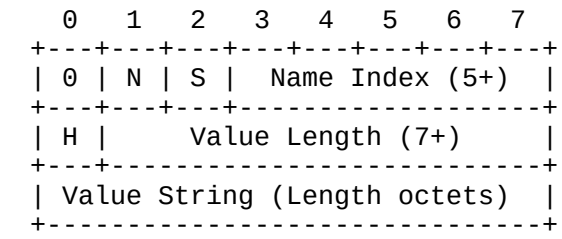
The index value of 0 is not used. It MUST be treated as a decoding error if found in an indexed header field representation.

3.3.2. Literal Header Field Representation

A literal header field representation starts with the ’0’ 1-bit pattern and causes a header field to be added the decoded header list.

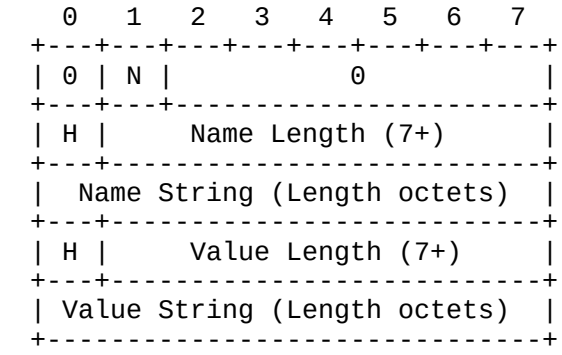
The second bit, ’N’, indicates whether an intermediary is permitted to add this header to the dynamic header table on subsequent hops. When the ’N’ bit is set, the encoded header MUST always be encoded with this specific literal representation. In particular, when a peer sends a header field that it received represented as a literal header field with the ’N’ bit set, it MUST use the same representation to forward this header field. This bit is intended for protecting header field values that are not to be put at risk by compressing them (see Section 7.1 of [RFC7541] for more details).

If the header field name matches the header field name of an entry stored in the static table or the dynamic table, the header field name can be represented using the index of that entry. In this case, the "S" bit indicates whether the reference is to the static (S=1) or dynamic (S=0) table and the index of the entry is represented as an integer with an 5-bit prefix (see Section 5.1 of [RFC7541]). This value is always non-zero.



Literal Header Field -- Indexed Name

Otherwise, the header field name is represented as a string literal (see Section 5.2 of [RFC7541]). A value 0 is used in place of the 6-bit index, followed by the header field name.



Literal Header Field -- Literal Name

Either form of header field name representation is followed by the header field value represented as a string literal (see Section 5.2 of [RFC7541]).

4. Use in HTTP/QUIC

HTTP/QUIC [I-D.ietf-quic-http] currently retains the HPACK encoder/decoder from HTTP/2, but restricts the size of the dynamic table to zero. Using QPACK instead would entail the following changes:

- Header Blocks consist of QPACK data instead of HPACK data
- HEADERS and PUSH_PROMISE frames define a flag indicating the presence of a preface.
- Just as unidirectional push streams have a stream header identifying their Push ID, a header will need to be added to differentiate checkpoint streams from pushes

- Stream 2 is reserved for the Feedback Stream

A HEADERS or PUSH_PROMISE frame MAY contain an arbitrary number of QPACK instructions. A partial HEADERS or PUSH_PROMISE frame MAY be processed upon arrival and the resulting partial header set emitted or buffered according to implementation requirements.

4.1. SETTING_QPACK_BLOCKING_PERMITTED

An HTTP/QUIC implementation can trade off the complexity of its QPACK decoder against compression efficiency by permitting the peer's compressor to reference unacknowledged entries. In the case of loss on a checkpoint stream, such references might cause the processing of request streams to block, waiting for the arrival of missing data.

If the decoder permits the encoder to make blocking references, it sets "SETTING_QPACK_BLOCKING_PERMITTED" (0xSETTING-TBD1) to a non-zero value. The encoder receiving this setting MAY encode up to this number of potentially-blocking references at a time.

Sending this setting with no value indicates that a decoder is willing to tolerate blocking references bounded only by the allowed number of streams. If a decoder does not send this setting or sends this setting with a value of zero, the encoder MUST NOT encode a header using a reference that might block.

4.2. SETTING_QPACK_INITIAL_CHECKPOINT

An HTTP/QUIC implementation MAY include the "SETTING_QPACK_INITIAL_CHECKPOINT" (0xSETTING_TBD2) setting, containing the full serialization of an initial checkpoint stream's data. If present, this setting MUST be fully processed by the peer before decoding any checkpoint streams or header frames on request streams.

The checkpoint defined by this setting is considered LIVE by both the encoder and the decoder from the beginning of the connection. The decoder does not need to send an ACK_FLUSH message confirming receipt of this setting.

5. Implementation trade-offs

This document specifies a means for the encoder to express the choices it made while encoding, but intentionally does not mandate what those choices should be. In this section, potential areas for implementation tuning are explored.

5.1. Compression Efficiency versus Blocking Avoidance

If blocking references are permitted, they will block if the frame containing the entry definition is lost or delayed. Encoders MAY choose to trade off compression efficiency and avoid blocking by using literal instructions rather than referencing the dynamic table until the insertion is believed to be complete.

The most efficient compression algorithm will reference a table entry whenever it exists in the table, but risks blocking when subject to packet loss or reordering. The most conservative algorithm will always emit literals to guarantee that no blocking will ever occur. Most implementations will choose a balance between these two extremes.

Better efficiency while being similarly conservative can be achieved by permitting references to table entries only once these entries are confirmed to be present in the table. More optimization can be achieved when the reference is known to be in the same packet as the definition.

Increases in efficiency can be achieved by assuming greater risk of blocking - implementations might choose a particular balance, or adjust their aggressiveness based on observed network characteristics.

Since it is possible to insert header values without emitting them on a stream, an encoder MAY also proactively insert header values which it believes will be needed on future requests, at the cost of reduced compression efficiency for incorrect predictions.

The ability to split updates to the header table into discrete checkpoints reduces the possibility for head-of-line blocking within the checkpoint streams. Implementations SHOULD limit the size of checkpoints to avoid head-of-line blocking within these messages.

5.2. Timely State Transitions versus Decoder Complexity

Anything which prevent checkpoints from transitioning from DYING to DEAD can prevent the encoder from adding any new entries due to the maximum table size. This does not block the encoder from continuing to make requests, but could sharply limit compression performance. Encoders would be well-served to begin moving checkpoint to DYING in advance of encountering the table maximum. Decoders SHOULD be prompt about emitting STREAM_DONE and ACK_DROP instructions to enable the encoder to recover the table space.

Similarly, for decoders which prohibit blocking references, delaying the transition of a checkpoint from PENDING to LIVE will degrade compression performance. Decoders SHOULD consume checkpoint data and emit ACK_FLUSH frames as promptly as possible.

Since decoders cannot safely drop old checkpoints until they have fully processed any checkpoints which might have been open concurrently, a long-lived checkpoint can delay the completion of an ACK_DROP. Encoders SHOULD flush all NEW checkpoints as soon as feasible after issuing a DROP instruction.

6. Security Considerations

A malicious encoder might attempt to consume a large amount of space on the decoder, but as each decoder chooses how much memory to allow the peer to consume, this state is bounded.

A malicious encoder might also send blocking references to entries which will never actually be defined. This attack is comparable to a "slow loris" attack in which a request is delivered very slowly in an attempt to consume resources on the server. Similar mitigations (request timers, etc.) SHOULD be employed to guard against such attacks.

7. IANA Considerations

This document registers two settings and one error code with the corresponding HTTP/QUIC registries.

7.1. Settings

This document registers two entries in the "HTTP/QUIC Settings" registry established by [I-D.ietf-quic-http].

Setting Name: SETTING_QPACK_BLOCKING_PERMITTED

Code: 0xSETTING-TBD1

Specification: Section 4.1

and

Setting Name: SETTING_QPACK_INITIAL_CHECKPOINT

Code: 0xSETTING-TBD2

Specification: Section 4.2

7.2. Errors

This document registers one error code in the "HTTP/QUIC Error Code" registry established by [I-D.ietf-quic-http].

Error name: ERROR_QPACK_INVALID_REFERENCE

Code: 0xERROR-TBD

Description: A blocking reference was received by a decoder which did not permit it

Specification: Section 2.2.1

8. Acknowledgements

This draft draws heavily on the text of [RFC7541], and adopts (with adaptation) the checkpoint model from [QMIN]. The direct and indirect input of those authors is gratefully acknowledged, as well as ideas gleefully stolen from:

- Jana Iyengar
- Patrick McManus
- Martin Thomson
- Charles 'Buck' Krasic
- Kyle Rose
- Alan Frindell

A substantial portion of Mike's work on this draft was supported by Microsoft during his employment there.

9. References

9.1. Normative References

[I-D.ietf-quic-http]
Bishop, M., "Hypertext Transfer Protocol (HTTP) over QUIC", draft-ietf-quic-http-07 (work in progress), October 2017.

[I-D.ietf-quic-transport]
Iyengar, J. and M. Thomson, "QUIC: A UDP-Based Multiplexed and Secure Transport", draft-ietf-quic-transport-07 (work in progress), October 2017.

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

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

[RFC7541] Peon, R. and H. Ruellan, "HPACK: Header Compression for HTTP/2", RFC 7541, DOI 10.17487/RFC7541, May 2015, <https://www.rfc-editor.org/info/rfc7541>.

9.2. Informative References

[QMIN] Tikhonov, D., "QMIN: Header Compression for QUIC", draft-tikhonov-quic-qmin-00 (work in progress), November 2017.

Author’s Address

Mike Bishop
Akamai

Email: mbishop@evequefou.be