QPACK: Header Compression for HTTP/3

draft-ietf-quic-qpack-09

Abstract

This specification defines QPACK, a compression format for efficiently representing HTTP header fields, to be used in HTTP/3. This is a variation of HPACK header compression that seeks to reduce head-of-line blocking.

Note to Readers

Discussion of this draft takes place on the QUIC working group mailing list (quic@ietf.org), which is archived at https://mailarchive.ietf.org/arch/search/?email_list=quic [1].

Working Group information can be found at https://github.com/quicwg [2]; source code and issues list for this draft can be found at https://github.com/quicwg/base-drafts/labels/-qpack [3].

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 January 9, 2020.
Copyright Notice

Copyright (c) 2019 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 (https://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 ......................................................... 3
   1.1. Conventions and Definitions .................................. 4
   1.2. Notational Conventions ....................................... 4
2. Compression Process Overview ...................................... 5
   2.1. Encoder ..................................................... 5
   2.1.1. Reference Tracking ....................................... 5
   2.1.2. Blocked Dynamic Table Insertions .......................... 6
   2.1.3. Blocked Streams .......................................... 7
   2.1.4. Known Received Count .................................... 8
   2.2. Decoder .................................................... 8
   2.2.1. Blocked Decoding ......................................... 8
   2.2.2. State Synchronization .................................... 9
   2.2.3. Invalid References ...................................... 10
3. Header Tables ..................................................... 10
   3.1. Static Table ................................................. 10
   3.2. Dynamic Table ............................................... 11
   3.2.1. Dynamic Table Size ....................................... 11
   3.2.2. Dynamic Table Capacity and Eviction ...................... 11
   3.2.3. Maximum Dynamic Table Capacity ........................... 12
   3.2.4. Absolute Indexing ....................................... 12
   3.2.5. Relative Indexing ....................................... 12
   3.2.6. Post-Base Indexing ..................................... 14
4. Wire Format ....................................................... 14
   4.1. Primitives .................................................. 14
   4.1.1. Prefixed Integers ....................................... 14
   4.1.2. String Literals ......................................... 14
   4.2. Encoder and Decoder Streams ................................ 15
   4.3. Encoder Instructions ....................................... 15
   4.3.1. Set Dynamic Table Capacity ............................... 16
   4.3.2. Insert With Name Reference ............................... 16
   4.3.3. Insert Without Name Reference ............................ 17
1. Introduction

The QUIC transport protocol [QUIC-TRANSPORT] is designed to support HTTP semantics, and its design subsumes many of the features of HTTP/2 [RFC7540]. HTTP/2 uses HPACK ([RFC7541]) for header compression. If HPACK were used for HTTP/3 [HTTP3], it would induce head-of-line blocking due to built-in assumptions of a total ordering across frames on all streams.
QPACK reuses core concepts from HPACK, but is redesigned to allow
correctness in the presence of out-of-order delivery, with
flexibility for implementations to balance between resilience against
head-of-line blocking and optimal compression ratio. The design
goals are to closely approach the compression ratio of HPACK with
substantially less head-of-line blocking under the same loss
conditions.

1.1. Conventions and Definitions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT",
"SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and
"OPTIONAL" in this document are to be interpreted as described in BCP
14 [RFC2119] [RFC8174] when, and only when, they appear in all
capitals, as shown here.

Definitions of terms that are used in this document:

Header field: A name-value pair sent as part of an HTTP message.

Header list: An ordered collection of header fields associated with
an HTTP message. A header list can contain multiple header fields
with the same name. It can also contain duplicate header fields.

Header block: The compressed representation of a header list.

Encoder: An implementation which transforms a header list into a
header block.

Decoder: An implementation which transforms a header block into a
header list.

Absolute Index: A unique index for each entry in the dynamic table.

Base: A reference point for relative indices. Dynamic references
are made relative to a Base in header blocks.

Insert Count: The total number of entries inserted in the dynamic
table.

QPACK is a name, not an acronym.

1.2. Notational Conventions

Diagrams use the format described in Section 3.1 of [RFC2360], with
the following additional conventions:

x (A) Indicates that x is A bits long
x (A+) Indicates that x uses the prefixed integer encoding defined in Section 5.1 of [RFC7541], beginning with an A-bit prefix.

x ... Indicates that x is variable-length and extends to the end of the region.

2. Compression Process Overview

Like HPACK, QPACK uses two tables for associating header fields to indices. The static table (see Section 3.1) is predefined and contains common header fields (some of them with an empty value). The dynamic table (see Section 3.2) is built up over the course of the connection and can be used by the encoder to index header fields in the encoded header lists.

QPACK defines unidirectional streams for sending instructions from encoder to decoder and vice versa.

2.1. Encoder

An encoder converts a header list into a header block by emitting either an indexed or a literal representation for each header field in the list (see Section 4.5). Indexed representations achieve high compression by replacing the literal name and possibly the value with an index to either the static or dynamic table. References to the static table and literal representations do not require any dynamic state and never risk head-of-line blocking. References to the dynamic table risk head-of-line blocking if the encoder has not received an acknowledgement indicating the entry is available at the decoder.

An encoder MAY insert any entry in the dynamic table it chooses; it is not limited to header fields it is compressing.

QPACK preserves the ordering of header fields within each header list. An encoder MUST emit header field representations in the order they appear in the input header list.

QPACK is designed to contain the more complex state tracking to the encoder, while the decoder is relatively simple.

2.1.1. Reference Tracking

An encoder MUST ensure that a header block which references a dynamic table entry is not processed by the decoder after the referenced entry has been evicted. Hence the encoder needs to track information about each compressed header block that references the dynamic table.
until that header block is acknowledged by the decoder (see Section 4.4.1).

2.1.2. Blocked Dynamic Table Insertions

A dynamic table entry is considered blocking and cannot be evicted until its insertion has been acknowledged and there are no outstanding unacknowledged references to the entry. In particular, a dynamic table entry that has never been referenced can still be blocking.

Note: A blocking entry is unrelated to a blocked stream, which is a stream that a decoder cannot decode as a result of references to entries that are not yet available. An encoder that uses the dynamic table has to keep track of blocked entries.

An encoder MUST NOT insert an entry into the dynamic table (or duplicate an existing entry) if doing so would evict a blocking entry.

2.1.2.1. Avoiding Blocked Insertions

To ensure that the encoder is not prevented from adding new entries, the encoder can avoid referencing entries that are close to eviction. Rather than reference such an entry, the encoder can emit a Duplicate instruction (see Section 4.3.4), and reference the duplicate instead.

Determining which entries are too close to eviction to reference is an encoder preference. One heuristic is to target a fixed amount of available space in the dynamic table: either unused space or space that can be reclaimed by evicting non-blocking entries. To achieve this, the encoder can maintain a draining index, which is the smallest absolute index (see Section 3.2.4) in the dynamic table that it will emit a reference for. As new entries are inserted, the encoder increases the draining index to maintain the section of the table that it will not reference. If the encoder does not create new references to entries with an absolute index lower than the draining index, the number of unacknowledged references to those entries will eventually become zero, allowing them to be evicted.
2.1.3. Blocked Streams

Because QUIC does not guarantee order between data on different streams, a decoder might encounter a header block that references a dynamic table entry that it has not yet received.

Each header block contains a Required Insert Count (see Section 4.5.1), the lowest possible value for the Insert Count with which the header block can be decoded. For a header block with references to the dynamic table, the Required Insert Count is one larger than the largest absolute index of all referenced dynamic table entries. For a header block with no references to the dynamic table, the Required Insert Count is zero.

When the decoder receives a header block with a Required Insert Count greater than its own Insert Count, the stream cannot be processed immediately, and is considered "blocked" (see (blocked-decoding)).

The SETTINGS_QPACK_BLOCKED_STREAMS setting (see Section 5) specifies an upper bound on the number of streams which can be blocked. An encoder MUST limit the number of streams which could become blocked to the value of SETTINGS_QPACK_BLOCKED_STREAMS at all times. Note that the decoder might not become blocked on every stream which risks becoming blocked.

An encoder can decide whether to risk having a stream become blocked. If permitted by the value of SETTINGS_QPACK_BLOCKED_STREAMS, compression efficiency can often be improved by referencing dynamic table entries that are still in transit, but if there is loss or reordering the stream can become blocked at the decoder. An encoder avoids the risk of blocking by only referencing dynamic table entries which have been acknowledged, but this could mean using literals. Since literals make the header block larger, this can result in the encoder becoming blocked on congestion or flow control limits.
2.1.4. Known Received Count

In order to identify which dynamic table entries can be safely used without a stream becoming blocked, the encoder tracks the number of entries received by the decoder. The Known Received Count tracks the total number of acknowledged insertions.

When blocking references are permitted, the encoder uses Header Acknowledgement instructions (Section 4.4.1) to maintain the Known Received Count. If a header block was potentially blocking, the acknowledgement implies that the decoder has received all dynamic table state necessary to process the header block. If the Required Insert Count of an acknowledged header block was greater than the encoder’s current Known Received Count, the block’s Required Insert Count becomes the new Known Received Count.

To acknowledge dynamic table entries which are not referenced by header blocks, for example because the encoder or the decoder have chosen not to risk blocked streams, the decoder sends an Insert Count Increment instruction (see Section 4.4.3).

2.2. Decoder

As in HPACK, the decoder processes header blocks and emits the corresponding header lists. It also processes dynamic table modifications from encoder instructions received on the encoder stream.

The decoder MUST emit header fields in the order their representations appear in the input header block.

2.2.1. Blocked Decoding

Upon receipt of a header block, the decoder examines the Required Insert Count. When the Required Insert Count is less than or equal to the decoder’s Insert Count, the header block can be processed immediately. Otherwise, the stream on which the header block was received becomes blocked.

While blocked, header block data SHOULD remain in the blocked stream’s flow control window. A stream becomes unblocked when the Insert Count becomes greater than or equal to the Required Insert Count for all header blocks the decoder has started reading from the stream.

When processing header blocks, the decoder expects the Required Insert Count to exactly match the value defined in {{blocked-streams}). If it encounters a smaller value than expected, it MUST
treat this as a connection error of type HTTP_QPACK_DECOMPRESSION_FAILED (see Section 2.2.3). If it encounters a larger value than expected, it MAY treat this as a connection error of type HTTP_QPACK_DECOMPRESSION_FAILED.

If the decoder encounters more blocked streams than it promised to support, it MUST treat this as a connection error of type HTTP_QPACK_DECOMPRESSION_FAILED.

2.2.2. State Synchronization

The decoder signals the following events by emitting decoder instructions (Section 4.4) on the decoder stream.

2.2.2.1. Completed Processing of a Header Block

When the decoder finishes decoding a header block containing dynamic table references, it emits a Header Acknowledgement instruction (Section 4.4.1). A stream may carry multiple header blocks in the case of intermediate responses, trailers, and pushed requests. The encoder interprets each Header Acknowledgement instruction as acknowledging the earliest unacknowledged header block containing dynamic table references sent on the given stream.

2.2.2.2. Abandonment of a Stream

When an endpoint receives a stream reset before the end of a stream or before all header blocks are processed on that stream, or when it abandons reading of a stream, it generates a Stream Cancellation instruction (see Section 4.4.2). This signals to the encoder that all references to the dynamic table on that stream are no longer outstanding. A decoder with a maximum dynamic table capacity equal to zero (see Section 3.2.3) MAY omit sending Stream Cancellations, because the encoder cannot have any dynamic table references. An encoder cannot infer from this instruction that any updates to the dynamic table have been received.

The Header Acknowledgement and Stream Cancellation instructions permit the encoder to remove references to entries in the dynamic table. When entries have zero references they are no longer considered blocking (see Section 2.1.2).

2.2.2.3. New Table Entries

After receiving new table entries on the encoder stream, the decoder chooses when to emit Insert Count Increment instructions (see Section 4.4.3). Emitting this instruction after adding each new dynamic table entry will provide the timeliest feedback to the
encoder, but could be redundant with other decoder feedback. By delaying an Insert Count Increment instruction, the decoder might be able to coalesce multiple Insert Count Increment instructions, or replace them entirely with Header Acknowledgements (see Section 4.4.1). However, delaying too long may lead to compression inefficiencies if the encoder waits for an entry to be acknowledged before using it.

2.2.3. Invalid References

If the decoder encounters a reference in a header block representation to a dynamic table entry which has already been evicted or which has an absolute index greater than or equal to the declared Required Insert Count (see Section 4.5.1), it MUST treat this as a connection error of type "HTTP_QPACK_DECOMPRESSION_FAILED".

If the decoder encounters a reference in an encoder instruction to a dynamic table entry which has already been evicted, it MUST treat this as a connection error of type "HTTP_QPACK_ENCODER_STREAM_ERROR".

3. Header Tables

Unlike in HPACK, entries in the QPACK static and dynamic tables are addressed separately. The following sections describe how entries in each table are addressed.

3.1. Static Table

The static table consists of a predefined static list of header fields, each of which has a fixed index over time. Its entries are defined in Appendix A.

All entries in the static table have a name and a value. However, values can be empty (that is, have a length of 0). Each entry is identified by a unique index.

Note the QPACK static table is indexed from 0, whereas the HPACK static table is indexed from 1.

When the decoder encounters an invalid static table index in a header block representation it MUST treat this as a connection error of type "HTTP_QPACK_DECOMPRESSION_FAILED". If this index is received on the encoder stream, this MUST be treated as a connection error of type "HTTP_QPACK_ENCODER_STREAM_ERROR".
### 3.2. Dynamic Table

The dynamic table consists of a list of header fields maintained in first-in, first-out order. Each HTTP/3 endpoint holds a dynamic table that is initially empty. Entries are added by encoder instructions received on the encoder stream (see Section 4.3).

The dynamic table can contain duplicate entries (i.e., entries with the same name and same value). Therefore, duplicate entries MUST NOT be treated as an error by the decoder.

Dynamic table entries can have empty values.

#### 3.2.1. Dynamic Table Size

The size of the dynamic table is the sum of the size of its entries.

The size of an entry is the sum of its name’s length in bytes (as defined in Section 4.1.2), its value’s length in bytes, and 32.

The size of an entry is calculated using the length of its name and value without Huffman encoding applied.

#### 3.2.2. Dynamic Table Capacity and Eviction

The encoder sets the capacity of the dynamic table, which serves as the upper limit on its size. The initial capacity of the dynamic table is zero. The encoder sends a Set Dynamic Table Capacity instruction (Section 4.3.1) with a non-zero capacity to begin using the dynamic table.

Before a new entry is added to the dynamic table, entries are evicted from the end of the dynamic table until the size of the dynamic table is less than or equal to (table capacity - size of new entry). The encoder MUST NOT evict a blocking dynamic table entry (see Section 2.1.2). The entry is then added to the table. It is an error if the encoder attempts to add an entry that is larger than the dynamic table capacity; the decoder MUST treat this as a connection error of type "HTTP_QPACK_ENCODER_STREAM_ERROR".

A new entry can reference an entry in the dynamic table that will be evicted when adding this new entry into the dynamic table. Implementations are cautioned to avoid deleting the referenced name or value if the referenced entry is evicted from the dynamic table prior to inserting the new entry.

Whenever the dynamic table capacity is reduced by the encoder (see Section 4.3.1), entries are evicted from the end of the dynamic table...
until the size of the dynamic table is less than or equal to the new
table capacity. This mechanism can be used to completely clear
entries from the dynamic table by setting a capacity of 0, which can
subsequently be restored.

3.2.3. Maximum Dynamic Table Capacity

To bound the memory requirements of the decoder, the decoder limits
the maximum value the encoder is permitted to set for the dynamic
table capacity. In HTTP/3, this limit is determined by the value of
SETTINGS_QPACK_MAX_TABLE_CAPACITY sent by the decoder (see
Section 5). The encoder MUST not set a dynamic table capacity that
exceeds this maximum, but it can choose to use a lower dynamic table
capacity (see Section 4.3.1).

For clients using 0-RTT data in HTTP/3, the server’s maximum table
capacity is the remembered value of the setting, or zero if the value
was not previously sent. When the client’s 0-RTT value of the
SETTING is 0, the server MAY set it to a non-zero value in its
SETTINGS frame. If the remembered value is non-zero, the server MUST
send the same non-zero value in its SETTINGS frame. If it specifies
any other value, or omits SETTINGS_QPACK_MAX_TABLE_CAPACITY from
SETTINGS, the encoder must treat this as a connection error of type
"HTTP_QPACK_DECODER_STREAM_ERROR".

For HTTP/3 servers and HTTP/3 clients when 0-RTT is not attempted or
is rejected, the maximum table capacity is 0 until the encoder
processes a SETTINGS frame with a non-zero value of
SETTINGS_QPACK_MAX_TABLE_CAPACITY.

When the maximum table capacity is 0, the encoder MUST NOT insert
entries into the dynamic table, and MUST NOT send any encoder
instructions on the encoder stream.

3.2.4. Absolute Indexing

Each entry possesses both an absolute index which is fixed for the
lifetime of that entry and a relative index which changes based on
the context of the reference. The first entry inserted has an
absolute index of "0"; indices increase by one with each insertion.

3.2.5. Relative Indexing

Relative indices begin at zero and increase in the opposite direction
from the absolute index. Determining which entry has a relative
index of "0" depends on the context of the reference.
In encoder instructions, a relative index of "0" always refers to the most recently inserted value in the dynamic table. Note that this means the entry referenced by a given relative index will change while interpreting instructions on the encoder stream.

```
+-----+---------------+-------+
| n-1 |      ...      |   d   |  Absolute Index
+ - - +---------------+ - - - +
|  0  |      ...      | n-d-1 |  Relative Index
+-----+---------------+-------+
```

Insertion Point               Dropping Point

n = count of entries inserted
d = count of entries dropped

Example Dynamic Table Indexing - Encoder Stream

Unlike encoder instructions, relative indices in header block representations are relative to the Base at the beginning of the header block (see Section 4.5.1). This ensures that references are stable even if header blocks and dynamic table updates are processed out of order.

In a header block a relative index of "0" refers to the entry with absolute index equal to Base - 1.

```
Required
Insert
Count        Base
|           |      |
| V         | V     |
```

```
+-----+-----+-----+-----+-------+
| n-1 | n-2 | n-3 | ... |   d   |  Absolute Index
+-----+-----+  -  +-----+   -   +
|  0  |     | n-d-3 |  Relative Index
+-----+-----+-------+
```

n = count of entries inserted
d = count of entries dropped

Example Dynamic Table Indexing - Relative Index in Header Block
3.2.6.  Post-Base Indexing

Post-Base indices are used for entries with absolute indexes greater
than or equal to Base, starting at 0 for the entry with absolute
index equal to Base, and increasing in the same direction as the
absolute index.

Post-Base indices allow an encoder to process a header block in a
single pass and include references to entries added while processing
this (or other) header blocks.

+-----+-----+-----+-----+-----+
| n-1 | n-2 | n-3 | ... | d  | Absolute Index
+-----+-----+-----+-----+-----+
|  1  |  0  |                    Post-Base Index
+-----+-----+

n = count of entries inserted
d = count of entries dropped

Example Dynamic Table Indexing - Post-Base Index in Header Block

4.  Wire Format

4.1.  Primitives

4.1.1.  Prefixed Integers

The prefixed integer from Section 5.1 of [RFC7541] is used heavily
throughout this document. The format from [RFC7541] is used
unmodified. QPACK implementations MUST be able to decode integers up
to 62 bits long.

4.1.2.  String Literals

The string literal defined by Section 5.2 of [RFC7541] is also used
throughout. This string format includes optional Huffman encoding.

HPACK defines string literals to begin on a byte boundary. They
begin with a single flag (indicating whether the string is Huffman-
coded), followed by the Length encoded as a 7-bit prefix integer, and
finally Length bytes of data. When Huffman encoding is enabled, the
Huffman table from Appendix B of [RFC7541] is used without
modification.
This document expands the definition of string literals and permits them to begin other than on a byte boundary. An "N-bit prefix string literal" begins with the same Huffman flag, followed by the length encoded as an (N-1)-bit prefix integer. The remainder of the string literal is unmodified.

A string literal without a prefix length noted is an 8-bit prefix string literal and follows the definitions in [RFC7541] without modification.

4.2. Encoder and Decoder Streams

QPACK defines two unidirectional stream types:

- An encoder stream is a unidirectional stream of type "0x02". It carries an unframed sequence of encoder instructions from encoder to decoder.
- A decoder stream is a unidirectional stream of type "0x03". It carries an unframed sequence of decoder instructions from decoder to encoder.

HTTP/3 endpoints contain a QPACK encoder and decoder. Each endpoint MUST initiate at most one encoder stream and at most one decoder stream. Receipt of a second instance of either stream type MUST be treated as a connection error of type HTTP_WRONG_STREAM_COUNT. These streams MUST NOT be closed. Closure of either unidirectional stream type MUST be treated as a connection error of type HTTP_CLOSED_CRITICAL_STREAM.

An endpoint MAY avoid creating its own encoder stream if it’s not going to be used (for example if the endpoint doesn’t wish to use the dynamic table, or if the maximum size of the dynamic table permitted by the peer is zero).

An endpoint MAY avoid creating its own decoder stream if the maximum size of its own dynamic table is zero.

An endpoint MUST allow its peer to create both encoder and decoder streams even if the connection’s settings prevent their use.

4.3. Encoder Instructions

An encoder sends encoder instructions on the encoder stream to set the capacity of the dynamic table and add dynamic table entries. Instructions adding table entries can use existing entries to avoid transmitting redundant information. The name can be transmitted as a reference to an existing entry in the static or the dynamic table or
as a string literal. For entries which already exist in the dynamic table, the full entry can also be used by reference, creating a duplicate entry.

This section specifies the following encoder instructions.

4.3.1. Set Dynamic Table Capacity

An encoder informs the decoder of a change to the dynamic table capacity using an instruction which begins with the ‘001’ three-bit pattern. The new dynamic table capacity is represented as an integer with a 5-bit prefix (see Section 5.1 of [RFC7541]).

```
0   1   2   3   4   5   6   7
+---+---+---+---+---+---+---+---+
| 0 | 0 | 1 |   Capacity (5+)   |
+---+---+---+-------------------+
```

Figure 2: Set Dynamic Table Capacity

The new capacity MUST be lower than or equal to the limit described in Section 3.2.3. In HTTP/3, this limit is the value of the SETTINGS_QPACK_MAX_TABLE_CAPACITY parameter (see Section 5) received from the decoder. The decoder MUST treat a new dynamic table capacity value that exceeds this limit as a connection error of type "HTTP_QPACK_ENCODER_STREAM_ERROR".

Reducing the dynamic table capacity can cause entries to be evicted (see Section 3.2.2). This MUST NOT cause the eviction of blocking entries (see Section 2.1.2). Changing the capacity of the dynamic table is not acknowledged as this instruction does not insert an entry.

4.3.2. Insert With Name Reference

An encoder adds an entry to the dynamic table where the header field name matches the header field name of an entry stored in the static or the dynamic table using an instruction that starts with the ‘1’ one-bit pattern. The second ("S") bit indicates whether the reference is to the static or dynamic table. The 6-bit prefix integer (see Section 5.1 of [RFC7541]) that follows is used to locate the table entry for the header name. When S=1, the number represents the static table index; when S=0, the number is the relative index of the entry in the dynamic table.

The header name reference is followed by the header field value represented as a string literal (see Section 5.2 of [RFC7541]).
4.3.3. Insert Without Name Reference

An encoder adds an entry to the dynamic table where both the header field name and the header field value are represented as string literals (see Section 4.1) using an instruction that starts with the ‘01’ two-bit pattern.

The name is represented as a 6-bit prefix string literal, while the value is represented as an 8-bit prefix string literal.

4.3.4. Duplicate

An encoder duplicates an existing entry in the dynamic table using an instruction that starts with the ‘000’ three-bit pattern. The relative index of the existing entry is represented as an integer with a 5-bit prefix.

Figure 3: Duplicate
The existing entry is re-inserted into the dynamic table without resending either the name or the value. This is useful to mitigate the eviction of older entries which are frequently referenced, both to avoid the need to resend the header and to avoid the entry in the table blocking the ability to insert new headers.

4.4. Decoder Instructions

Decoder instructions provide information used to ensure consistency of the dynamic table. They are sent from the decoder to the encoder on a decoder stream; that is, the server informs the client about the processing of the client’s header blocks and table updates, and the client informs the server about the processing of the server’s header blocks and table updates.

This section specifies the following decoder instructions.

4.4.1. Header Acknowledgement

After processing a header block whose declared Required Insert Count is not zero, the decoder emits a Header Acknowledgement instruction. The instruction begins with the ‘1’ one-bit pattern and includes the header block’s associated stream ID, encoded as a 7-bit prefix integer. It is used by the peer’s encoder to know when it is safe to evict an entry (Section 2.1.2), and possibly update the Known Received Count (Section 2.1.4).

```
+---+---+---+---+---+---+---+---+
| 1 | Stream ID (7+) |
+---+---------------------------+
```

Figure 4: Header Acknowledgement

If an encoder receives a Header Acknowledgement instruction referring to a stream on which every header block with a non-zero Required Insert Count has already been acknowledged, that MUST be treated as a connection error of type "HTTP_QPACK_DECODER_STREAM_ERROR".

4.4.2. Stream Cancellation

When a stream is reset or reading is abandoned, the decoder emits a Stream Cancellation instruction. The instruction begins with the ‘01’ two-bit pattern. The instruction includes the stream ID of the affected stream encoded as a 6-bit prefix integer. See Section 2.2.2.
4.4.3. Insert Count Increment

The Insert Count Increment instruction begins with the ’00’ two-bit pattern. The instruction specifies the total number of dynamic table inserts and duplications since the last Insert Count Increment or Header Acknowledgement that increased the Known Received Count for the dynamic table (see Section 2.1.4). The Increment field is encoded as a 6-bit prefix integer. The encoder uses this value to determine which table entries might cause a stream to become blocked, as described in Section 2.2.2.

An encoder that receives an Increment field equal to zero or one that increases the Known Received Count beyond what the encoder has sent MUST treat this as a connection error of type "HTTP_QPACK_DECODER_STREAM_ERROR".

4.5. Header Block Representations

Header blocks contain compressed representations of header lists and are carried in frames on streams defined by the enclosing protocol. These representations reference the static table, or dynamic table in a particular state without modifying it.

4.5.1. Header Block Prefix

Each header block is prefixed with two integers. The Required Insert Count is encoded as an integer with an 8-bit prefix after the encoding described in Section 4.5.1.1). The Base is encoded as sign- and-modulus integer, using a single sign bit and a value with a 7-bit prefix (see Section 4.5.1.2).

These two values are followed by representations for compressed headers.
4.5.1.1. Required Insert Count

Required Insert Count identifies the state of the dynamic table needed to process the header block. Blocking decoders use the Required Insert Count to determine when it is safe to process the rest of the block.

The encoder transforms the Required Insert Count as follows before encoding:

    if ReqInsertCount == 0:
        EncInsertCount = 0
    else:
        EncInsertCount = (ReqInsertCount mod (2 * MaxEntries)) + 1

Here "MaxEntries" is the maximum number of entries that the dynamic table can have. The smallest entry has empty name and value strings and has the size of 32. Hence "MaxEntries" is calculated as

    MaxEntries = floor( MaxTableCapacity / 32 )

"MaxTableCapacity" is the maximum capacity of the dynamic table as specified by the decoder (see Section 3.2.3).

This encoding limits the length of the prefix on long-lived connections.

The decoder can reconstruct the Required Insert Count using an algorithm such as the following. If the decoder encounters a value of EncodedInsertCount that could not have been produced by a conformant encoder, it MUST treat this as a connection error of type "HTTP_QPACK_DECOMPRESSION_FAILED".

TotalNumberOfInserts is the total number of inserts into the decoder’s dynamic table.
FullRange = 2 * MaxEntries
if EncodedInsertCount == 0:
    ReqInsertCount = 0
else:
    if EncodedInsertCount > FullRange:
        Error
    MaxValue = TotalNumberOfInserts + MaxEntries

    # MaxWrapped is the largest possible value of
    # ReqInsertCount that is 0 mod 2*MaxEntries
    MaxWrapped = floor(MaxValue / FullRange) * FullRange
    ReqInsertCount = MaxWrapped + EncodedInsertCount - 1

    # If ReqInsertCount exceeds MaxValue, the Encoder’s value
    # must have wrapped one fewer time
    if ReqInsertCount > MaxValue:
        if ReqInsertCount < FullRange:
            Error
        ReqInsertCount -= FullRange

For example, if the dynamic table is 100 bytes, then the Required
Insert Count will be encoded modulo 6. If a decoder has received 10
inserts, then an encoded value of 3 indicates that the Required
Insert Count is 9 for the header block.

4.5.1.2. Base

The "Base" is used to resolve references in the dynamic table as
described in Section 3.2.5.

To save space, the Base is encoded relative to the Insert Count using
a one-bit sign and the "Delta Base" value. A sign bit of 0 indicates
that the Base is greater than or equal to the value of the Insert
Count; the value of Delta Base is added to the Insert Count to
determine the value of the Base. A sign bit of 1 indicates that the
Base is less than the Insert Count. That is:

    if S == 0:
        Base = ReqInsertCount + DeltaBase
    else:
        Base = ReqInsertCount - DeltaBase - 1

A single-pass encoder determines the Base before encoding a header
block. If the encoder inserted entries in the dynamic table while
encoding the header block, Required Insert Count will be greater than
the Base, so the encoded difference is negative and the sign bit is
set to 1. If the header block did not reference the most recent
entry in the table and did not insert any new entries, the Base will
be greater than the Required Insert Count, so the delta will be positive and the sign bit is set to 0.

An encoder that produces table updates before encoding a header block might set Required Insert Count and the Base to the same value. In such case, both the sign bit and the Delta Base will be set to zero.

A header block that does not reference the dynamic table can use any value for the Base; setting Delta Base to zero is the most efficient encoding.

For example, with a Required Insert Count of 9, a decoder receives a S bit of 1 and a Delta Base of 2. This sets the Base to 6 and enables post-base indexing for three entries. In this example, a regular index of 1 refers to the 5th entry that was added to the table; a post-base index of 1 refers to the 8th entry.

4.5.2. Indexed Header Field

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].

```
  0 1 2 3 4 5 6 7
+---+---+---+---+---+---+---+---+
| 1 | S |      Index (6+)       |
+---+---+-----------------------+
```

Indexed Header Field

If the entry is in the static table, or in the dynamic table with an absolute index less than the Base, this representation starts with the '1' 1-bit pattern, followed by the "S" bit indicating whether the reference is into the static or dynamic table. The 6-bit prefix integer (see Section 5.1 of [RFC7541]) that follows is used to locate the table entry for the header name. When S=1, the number represents the static table index; when S=0, the number is the relative index of the entry in the dynamic table.

4.5.3. Indexed Header Field With Post-Base Index

If the entry is in the dynamic table with an absolute index greater than or equal to the Base, the representation starts with the '0001' 4-bit pattern, followed by the post-base index (see Section 3.2.6) of the matching header field, represented as an integer with a 4-bit prefix (see Section 5.1 of [RFC7541]).
4.5.4. Literal Header Field With Name Reference

A literal header field with a name reference represents a header where the header field name matches the header field name of an entry stored in the static table or the dynamic table.

If the entry is in the static table, or in the dynamic table with an absolute index less than the Base, this representation starts with the ‘01’ two-bit pattern.

Only the header field name stored in the static or dynamic table is used. Any header field value MUST be ignored.

The following 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 a 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 a literal 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).

The fourth ("S") bit indicates whether the reference is to the static or dynamic table. The 4-bit prefix integer (see Section 5.1 of [RFC7541]) that follows is used to locate the table entry for the header name. When S=1, the number represents the static table index; when S=0, the number is the relative index of the entry in the dynamic table.
4.5.5. Literal Header Field With Post-Base Name Reference

If the name entry is in the dynamic table with an absolute index greater than or equal to the Base, the representation starts with the ‘0000’ four-bit pattern. The fifth bit is the ‘N’ bit as described in Section 4.5.4. Finally, the header field name is represented using the post-base index of that entry (see Section 3.2.6) encoded as an integer with a 3-bit prefix.

```
0   1   2   3   4   5   6   7
+---+---+---+---+---+---+---+---+
| 0 | 0 | 0 | N | NameIdx(3+) |
+---+---+---+---+-----------+
| H | Value Length (7+)     |
+--------------------------------+
| Value String (Length bytes)  |
+-------------------------------+
```

Literal Header Field With Post-Base Name Reference

4.5.6. Literal Header Field Without Name Reference

An addition to the header table where both the header field name and the header field value are represented as string literals (see Section 4.1) starts with the ‘001’ three-bit pattern.

The fourth 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 a 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 a literal 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).

The name is represented as a 4-bit prefix string literal, while the value is represented as an 8-bit prefix string literal.
5. Configuration

QPACK defines two settings which are included in the HTTP/3 SETTINGS frame.

SETTINGS_QPACK_MAX_TABLE_CAPACITY (0x1): The default value is zero. See Section 3.2 for usage. This is the equivalent of the SETTINGS_HEADER_TABLE_SIZE from HTTP/2.

SETTINGS_QPACK_BLOCKED_STREAMS (0x7): The default value is zero. See Section 2.1.3.

6. Error Handling

The following error codes are defined for HTTP/3 to indicate failures of QPACK which prevent the connection from continuing:

HTTP_QPACK_DECOMPRESSION_FAILED (0x200): The decoder failed to interpret a header block and is not able to continue decoding that header block.

HTTP_QPACK_ENCODER_STREAM_ERROR (0x201): The decoder failed to interpret an encoder instruction received on the encoder stream.

HTTP_QPACK_DECODER_STREAM_ERROR (0x202): The encoder failed to interpret a decoder instruction received on the decoder stream.

7. Security Considerations

TBD.
8. IANA Considerations

8.1. Settings Registration

This document specifies two settings. The entries in the following table are registered in the "HTTP/3 Settings" registry established in [HTTP3].

<table>
<thead>
<tr>
<th>Setting Name</th>
<th>Code</th>
<th>Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>QPACK_MAX_TABLE_CAPACITY</td>
<td>0x1</td>
<td>Section 5</td>
</tr>
<tr>
<td>QPACK_BLOCKED_STREAMS</td>
<td>0x7</td>
<td>Section 5</td>
</tr>
</tbody>
</table>

8.2. Stream Type Registration

This document specifies two stream types. The entries in the following table are registered in the "HTTP/3 Stream Type" registry established in [HTTP3].

<table>
<thead>
<tr>
<th>Stream Type</th>
<th>Code</th>
<th>Specification</th>
<th>Sender</th>
</tr>
</thead>
<tbody>
<tr>
<td>QPACK Encoder Stream</td>
<td>0x02</td>
<td>Section 4.2</td>
<td>Both</td>
</tr>
<tr>
<td>QPACK Decoder Stream</td>
<td>0x03</td>
<td>Section 4.2</td>
<td>Both</td>
</tr>
</tbody>
</table>

8.3. Error Code Registration

This document specifies three error codes. The entries in the following table are registered in the "HTTP/3 Error Code" registry established in [HTTP3].
<table>
<thead>
<tr>
<th>Name</th>
<th>Code</th>
<th>Description</th>
<th>Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>HTTP_QPACK_DECOMPRESSION_FAILED</td>
<td>0x200</td>
<td>Decompression of a header block failed</td>
<td>Section 6</td>
</tr>
<tr>
<td>HTTP_QPACK_ENCODER_STREAM_ERROR</td>
<td>0x201</td>
<td>Error on the encoder stream</td>
<td>Section 6</td>
</tr>
<tr>
<td>HTTP_QPACK_DECODER_STREAM_ERROR</td>
<td>0x202</td>
<td>Error on the decoder stream</td>
<td>Section 6</td>
</tr>
</tbody>
</table>

9. References

9.1. Normative References


9.2. Informative References


9.3. URIs

[1] https://mailarchive.ietf.org/arch/search/?email_list=quic


Appendix A. Static Table

<table>
<thead>
<tr>
<th>Inde</th>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>:authority</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>:path</td>
<td>/</td>
</tr>
<tr>
<td>1</td>
<td>age</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>content-disposition</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>content-length</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>cookie</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>date</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>etag</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>if-modified-since</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>if-none-match</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>last-modified</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>link</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>location</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>referer</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>set-cookie</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>:method</td>
<td>CONNECT</td>
</tr>
<tr>
<td>16</td>
<td>:method</td>
<td>DELETE</td>
</tr>
<tr>
<td>17</td>
<td>:method</td>
<td>GET</td>
</tr>
<tr>
<td>18</td>
<td>:method</td>
<td>HEAD</td>
</tr>
<tr>
<td>19</td>
<td>:method</td>
<td>OPTIONS</td>
</tr>
<tr>
<td>20</td>
<td>:method</td>
<td>POST</td>
</tr>
<tr>
<td>21</td>
<td>:method</td>
<td>PUT</td>
</tr>
<tr>
<td>22</td>
<td>:scheme</td>
<td>http</td>
</tr>
<tr>
<td>23</td>
<td>:scheme</td>
<td>https</td>
</tr>
<tr>
<td>24</td>
<td>:status</td>
<td>103</td>
</tr>
<tr>
<td>25</td>
<td>:status</td>
<td>200</td>
</tr>
<tr>
<td>26</td>
<td>:status</td>
<td>304</td>
</tr>
<tr>
<td>27</td>
<td>:status</td>
<td>404</td>
</tr>
<tr>
<td>28</td>
<td>:status</td>
<td>503</td>
</tr>
<tr>
<td>29</td>
<td>accept</td>
<td><em>/</em></td>
</tr>
<tr>
<td>30</td>
<td>accept</td>
<td>application/dns-message</td>
</tr>
<tr>
<td>31</td>
<td>accept-encoding</td>
<td>gzip, deflate, br</td>
</tr>
<tr>
<td>32</td>
<td>accept-ranges</td>
<td>bytes</td>
</tr>
<tr>
<td>33</td>
<td>access-control-allow-headers</td>
<td>cache-control</td>
</tr>
<tr>
<td>34</td>
<td>access-control-allow-headers</td>
<td>content-type</td>
</tr>
<tr>
<td>35</td>
<td>access-control-allow-origin</td>
<td>*</td>
</tr>
<tr>
<td>36</td>
<td>cache-control</td>
<td>max-age=0</td>
</tr>
<tr>
<td>37</td>
<td>cache-control</td>
<td>max-age=2592000</td>
</tr>
<tr>
<td>38</td>
<td>cache-control</td>
<td>max-age=604800</td>
</tr>
<tr>
<td>39</td>
<td>cache-control</td>
<td>no-cache</td>
</tr>
<tr>
<td>40</td>
<td>cache-control</td>
<td>no-store</td>
</tr>
<tr>
<td>41</td>
<td>cache-control</td>
<td>public, max-age=31536000</td>
</tr>
<tr>
<td>42</td>
<td>content-encoding</td>
<td>br</td>
</tr>
<tr>
<td>43</td>
<td>content-encoding</td>
<td>gzip</td>
</tr>
<tr>
<td>44</td>
<td>content-type</td>
<td>application/dns-message</td>
</tr>
<tr>
<td>45</td>
<td>content-type</td>
<td>application/javascript</td>
</tr>
<tr>
<td>46</td>
<td>content-type</td>
<td>application/json</td>
</tr>
<tr>
<td>47</td>
<td>content-type</td>
<td>application/x-www-form-urlencoded</td>
</tr>
<tr>
<td>48</td>
<td>content-type</td>
<td>image/gif</td>
</tr>
<tr>
<td>49</td>
<td>content-type</td>
<td>image/jpeg</td>
</tr>
<tr>
<td>50</td>
<td>content-type</td>
<td>image/png</td>
</tr>
<tr>
<td>51</td>
<td>content-type</td>
<td>text/css</td>
</tr>
<tr>
<td>52</td>
<td>content-type</td>
<td>text/html; charset=utf-8</td>
</tr>
<tr>
<td>53</td>
<td>content-type</td>
<td>text/plain</td>
</tr>
<tr>
<td>54</td>
<td>content-type</td>
<td>text/plain; charset=utf-8</td>
</tr>
<tr>
<td>55</td>
<td>range</td>
<td>bytes=0-</td>
</tr>
<tr>
<td>56</td>
<td>strict-transport-security</td>
<td>max-age=31536000</td>
</tr>
<tr>
<td>57</td>
<td>strict-transport-security</td>
<td>max-age=31536000; includesubdomains</td>
</tr>
<tr>
<td>Line</td>
<td>Header</td>
<td>Value</td>
</tr>
<tr>
<td>------</td>
<td>------------------------------</td>
<td>--------------------------------------------</td>
</tr>
<tr>
<td>58</td>
<td>strict-transport-security</td>
<td>max-age=31536000; includesubdomains; preload</td>
</tr>
<tr>
<td>59</td>
<td>vary</td>
<td>accept-encoding</td>
</tr>
<tr>
<td>60</td>
<td>vary</td>
<td>origin</td>
</tr>
<tr>
<td>61</td>
<td>x-content-type-options</td>
<td>nosniff</td>
</tr>
<tr>
<td>62</td>
<td>x-xss-protection</td>
<td>1; mode=block</td>
</tr>
<tr>
<td>63</td>
<td>:status</td>
<td>100</td>
</tr>
<tr>
<td>64</td>
<td>:status</td>
<td>204</td>
</tr>
<tr>
<td>65</td>
<td>:status</td>
<td>206</td>
</tr>
<tr>
<td>66</td>
<td>:status</td>
<td>302</td>
</tr>
<tr>
<td>67</td>
<td>:status</td>
<td>400</td>
</tr>
<tr>
<td>68</td>
<td>:status</td>
<td>403</td>
</tr>
<tr>
<td>69</td>
<td>:status</td>
<td>421</td>
</tr>
<tr>
<td>70</td>
<td>:status</td>
<td>425</td>
</tr>
<tr>
<td>71</td>
<td>:status</td>
<td>500</td>
</tr>
<tr>
<td>72</td>
<td>accept-language</td>
<td></td>
</tr>
<tr>
<td>73</td>
<td>access-control-allow-</td>
<td>FALSE</td>
</tr>
<tr>
<td></td>
<td>credentials</td>
<td></td>
</tr>
<tr>
<td>74</td>
<td>access-control-allow-</td>
<td>TRUE</td>
</tr>
<tr>
<td></td>
<td>credentials</td>
<td></td>
</tr>
<tr>
<td>75</td>
<td>access-control-allow-</td>
<td>*</td>
</tr>
<tr>
<td></td>
<td>headers</td>
<td></td>
</tr>
<tr>
<td>76</td>
<td>access-control-allow-</td>
<td>get</td>
</tr>
<tr>
<td></td>
<td>methods</td>
<td></td>
</tr>
<tr>
<td>77</td>
<td>access-control-allow-</td>
<td>get, post, options</td>
</tr>
<tr>
<td></td>
<td>methods</td>
<td></td>
</tr>
<tr>
<td>78</td>
<td>access-control-allow-</td>
<td>options</td>
</tr>
<tr>
<td></td>
<td>methods</td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>----------------------------------</td>
<td>------------------------------</td>
</tr>
<tr>
<td>79</td>
<td>access-control-expose-headers</td>
<td>content-length</td>
</tr>
<tr>
<td>80</td>
<td>access-control-request-headers</td>
<td>content-type</td>
</tr>
<tr>
<td>81</td>
<td>access-control-request-method</td>
<td>get</td>
</tr>
<tr>
<td>82</td>
<td>access-control-request-method</td>
<td>post</td>
</tr>
<tr>
<td>83</td>
<td>alt-svc</td>
<td>clear</td>
</tr>
<tr>
<td>84</td>
<td>authorization</td>
<td></td>
</tr>
<tr>
<td>85</td>
<td>content-security-policy</td>
<td>script-src ‘none’; object-src ‘none’; base-uri ‘none’</td>
</tr>
<tr>
<td>86</td>
<td>early-data</td>
<td>1</td>
</tr>
<tr>
<td>87</td>
<td>expect-ct</td>
<td></td>
</tr>
<tr>
<td>88</td>
<td>forwarded</td>
<td></td>
</tr>
<tr>
<td>89</td>
<td>if-range</td>
<td></td>
</tr>
<tr>
<td>90</td>
<td>origin</td>
<td></td>
</tr>
<tr>
<td>91</td>
<td>purpose</td>
<td>prefetch</td>
</tr>
<tr>
<td>92</td>
<td>server</td>
<td></td>
</tr>
<tr>
<td>93</td>
<td>timing-allow-origin</td>
<td>*</td>
</tr>
<tr>
<td>94</td>
<td>upgrade-insecure-requests</td>
<td>1</td>
</tr>
<tr>
<td>95</td>
<td>user-agent</td>
<td></td>
</tr>
<tr>
<td>96</td>
<td>x-forwarded-for</td>
<td></td>
</tr>
<tr>
<td>97</td>
<td>x-frame-options</td>
<td>deny</td>
</tr>
<tr>
<td>98</td>
<td>x-frame-options</td>
<td>sameorigin</td>
</tr>
</tbody>
</table>
Appendix B. Sample One Pass Encoding Algorithm

Pseudo-code for single pass encoding, excluding handling of duplicates, non-blocking mode, and reference tracking.
baseIndex = dynamicTable.baseIndex
largestReference = 0
for header in headers:
    staticIdx = staticTable.getIndex(header)
    if staticIdx:
        encodeIndexReference(streamBuffer, staticIdx)
        continue

dynamicIdx = dynamicTable.getIndex(header)
if !dynamicIdx:
    # No matching entry. Either insert+index or encode literal
    nameIdx = getNameIndex(header)
    if shouldIndex(header) and dynamicTable.canIndex(header):
        encodeLiteralWithIncrementalIndex(controlBuffer, nameIdx, header)
    dynamicTable.add(header)
    dynamicIdx = dynamicTable.baseIndex
if !dynamicIdx:
    # Couldn’t index it, literal
    if nameIdx <= staticTable.size:
        encodeLiteral(streamBuffer, nameIndex, header)
    else:
        # encode literal, possibly with nameIdx above baseIndex
        encodeDynamicLiteral(streamBuffer, nameIndex, baseIndex, header)
        largestReference = max(largestReference,
                               dynamicTable.toAbsolute(nameIdx))
else:
    # Dynamic index reference
    assert(dynamicIdx)
    largestReference = max(largestReference, dynamicIdx)
    # Encode dynamic index, possibly with dynamicIdx above baseIndex
    encodeDynamicIndexReference(streamBuffer, dynamicIdx, baseIndex)

# encode the prefix
encodeInteger(prefixBuffer, 0x00, largestReference, 8)
if baseIndex >= largestReference:
    encodeInteger(prefixBuffer, 0, baseIndex - largestReference, 7)
else:
    encodeInteger(prefixBuffer, 0x80, largestReference - baseIndex, 7)

return controlBuffer, prefixBuffer + streamBuffer
Appendix C. Change Log

*RFC Editor’s Note:* Please remove this section prior to publication of a final version of this document.

C.1. Since draft-ietf-quic-qpack-08

- Endpoints are permitted to create encoder and decoder streams even if they can’t use them (#2100, #2529)
- Maximum values for settings removed (#2766, #2767)

C.2. Since draft-ietf-quic-qpack-06

- Clarify initial dynamic table capacity maximums (#2276, #2330, #2330)

C.3. Since draft-ietf-quic-qpack-05

- Introduced the terms dynamic table capacity and maximum dynamic table capacity.
- Renamed SETTINGS_HEADER_TABLE_SIZE to SETTINGS_QPACK_MAX_TABLE_CAPACITY.

C.4. Since draft-ietf-quic-qpack-04

- Changed calculation of Delta Base Index to avoid an illegal value (#2002, #2005)

C.5. Since draft-ietf-quic-qpack-03

- Change HTTP settings defaults (#2038)
- Substantial editorial reorganization

C.6. Since draft-ietf-quic-qpack-02

- Largest Reference encoded modulo MaxEntries (#1763)
- New Static Table (#1355)
- Table Size Update with Insert Count=0 is a connection error (#1762)
- Stream Cancellations are optional when SETTINGS_HEADER_TABLE_SIZE=0 (#1761)
o Implementations must handle 62 bit integers (#1760)

o Different error types for each QPACK stream, other changes to error handling (#1726)

o Preserve header field order (#1725)

o Initial table size is the maximum permitted when table is first usable (#1642)

C.7. Since draft-ietf-quic-qpack-01

o Only header blocks that reference the dynamic table are acknowledged (#1603, #1605)

C.8. Since draft-ietf-quic-qpack-00

o Renumbered instructions for consistency (#1471, #1472)

o Decoder is allowed to validate largest reference (#1404, #1469)

o Header block acknowledgments also acknowledge the associated largest reference (#1370, #1400)

o Added an acknowledgment for unread streams (#1371, #1400)

o Removed framing from encoder stream (#1361, #1467)

o Control streams use typed unidirectional streams rather than fixed stream IDs (#910, #1359)

C.9. Since draft-ietf-quic-qcram-00

o Separate instruction sets for table updates and header blocks (#1235, #1142, #1141)

o Reworked indexing scheme (#1176, #1145, #1136, #1130, #1125, #1314)

o Added mechanisms that support one-pass encoding (#1138, #1320)

o Added a setting to control the number of blocked decoders (#238, #1140, #1143)

o Moved table updates and acknowledgments to dedicated streams (#1121, #1122, #1238)
Acknowledgments

This draft draws heavily on the text of [RFC7541]. The indirect input of those authors is gratefully acknowledged, as well as ideas from:

- Ryan Hamilton
- Patrick McManus
- Kazuho Oku
- Biren Roy
- Ian Swett
- Dmitri Tikhonov

Buck’s contribution was supported by Google during his employment there.

A substantial portion of Mike’s contribution was supported by Microsoft during his employment there.

Authors’ Addresses

Charles ‘Buck’ Krasic
Netflix
Email: ckrasic@netflix.com

Mike Bishop
Akamai Technologies
Email: mbishop@evequefou.be

Alan Frindell (editor)
Facebook
Email: afrind@fb.com