Workgroup: QUIC
Internet-Draft:
draft-ietf-quic-qlog-h3-events-03
Published: 24 October 2022
Intended Status: Standards Track
Expires: 27 April 2023
Authors: R. Marx, Ed. L. Niccolini, Ed. M. Seemann, Ed.
Akamai Meta Protocol Labs
L. Pardue, Ed.
Cloudflare

HTTP/3 and QPACK qlog event definitions

Abstract

This document describes concrete qlog event definitions and their metadata for HTTP/3 and QPACK-related events. These events can then be embedded in the higher level schema defined in [QLOG-MAIN].

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

This document describes the values of the qlog name ("category" + "event") and "data" fields and their semantics for HTTP/3 [HTTP/3] and QPACK [QPACK].

Note to RFC editor: Please remove the follow paragraphs in this section before publication.

Feedback and discussion are welcome at https://github.com/quicwg/qlog. Readers are advised to refer to the "editor's draft" at that URL for an up-to-date version of this document.

Concrete examples of integrations of this schema in various programming languages can be found at https://github.com/quiclog/qlog/.

1.1. Notational Conventions

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.

The event and data structure definitions in ths document are expressed in the Concise Data Definition Language [CDDL] and its extensions described in [QLOG-MAIN].

The following fields from [QLOG-MAIN] are imported and used: name, category, type, data, group_id, protocol_type, importance, RawInfo, and time-related fields.

2. Overview

This document describes how the HTTP/3 and QPACK can be expressed in qlog using the schema defined in [QLOG-MAIN]. HTTP/3 and QPACK events are defined with a category, a name (the concatenation of "category" and "event"), an "importance", an optional "trigger", and "data" fields.

Some data fields use complex datastructures. These are represented as enums or re-usable definitions, which are grouped together on the bottom of this document for clarity.

When any event from this document is included in a qlog trace, the "protocol_type" qlog array field **MUST** contain an entry with the value "HTTP3".

2.1. Usage with QUIC

The events described in this document can be used with or without logging the related QUIC events defined in [QLOG-QUIC]. If used with QUIC events, the QUIC document takes precedence in terms of recommended filenames and trace separation setups.

If used without QUIC events, it is recommended that the implementation assign a globally unique identifier to each HTTP/3 connection. This ID can then be used as the value of the qlog "group_id" field, as well as the qlog filename or file identifier, potentially suffixed by the vantagepoint type (For example, abcd1234_server.qlog would contain the server-side trace of the connection with GUID abcd1234).

2.2. Raw packet and frame information

This document re-uses the definition of the RawInfo data class from [QLOG-MAIN].

Note: As HTTP/3 does not use trailers in frames, each HTTP/3 frame header_length can be calculated as header_length = RawInfo:length - RawInfo:payload_length

Note: In some cases, the length fields are also explicitly reflected inside of frame headers. For example, all HTTP/3 frames include their explicit payload lengths in the frame header. In these cases, those fields are intentionally preserved in the event definitions. Even though this can lead to duplicate data when the full RawInfo is logged, it allows a more direct mapping of the HTTP/3 specifications to qlog, making it easier for users to interpret. In this case, both fields **MUST** have the same value.

3. HTTP/3 and QPACK Event Overview

This document defines events in two categories, written as lowercase to follow convention: http (Section 4) and qpack (Section 6).

As described in Section 3.4.2 of [QLOG-MAIN], the qlog "name" field is the concatenation of category and type.

Table 1 summarizes the name value of each event type that is defined in this specification.

Name value	Importance	Definition
http:parameters_set	Base	Section 4.1
http:parameters_restored	Base	Section 4.2
http:stream_type_set	Base	Section 4.3
http:frame_created	Core	Section 4.4
http:frame_parsed	Core	Section 4.5
http:push_resolved	Extra	Section 4.6
qpack:state_updated	Base	Section 6.1
qpack:stream_state_updated	Core	Section 6.2
qpack:dynamic_table_updated	Extra	Section 6.3
qpack:headers_encoded	Base	Section 6.4
qpack:headers_decoded	Base	Section 6.5
qpack:instruction_created	Base	Section 6.6
qpack:instruction_parsed	Base	Section 6.7

Table 1: HTTP/3 and QPACK Events

4. HTTP/3 Events

HTTP/3 events extend the $ProtocolEventBody extension point defined in [QLOG-MAIN].

```
HTTPEvents = HTTPParametersSet / HTTPParametersRestored /
             HTTPStreamTypeSet / HTTPFrameCreated /
             HTTPFrameParsed / HTTPPushResolved

$ProtocolEventBody /= HTTPEvents
```

Figure 1: HTTPEvents definition and ProtocolEventBody extension

4.1. parameters_set

Importance: Base

This event contains HTTP/3 and QPACK-level settings, mostly those received from the HTTP/3 SETTINGS frame. All these parameters are typically set once and never change. However, they are typically set at different times during the connection, so there can be several instances of this event with different fields set.

Note that some settings have two variations (one set locally, one requested by the remote peer). This is reflected in the "owner" field. As such, this field **MUST** be correct for all settings included a single event instance. If you need to log settings from two sides, you **MUST** emit two separate event instances.

Note: we use the CDDL unwrap operator (~) here to make HTTPParameters into a re-usable list of fields. The unwrap operator copies the fields from the referenced type into the target type directly, extending the target with the unwrapped fields. TODO: explain this better + provide reference and maybe an example.

Definition:

```
HTTPParametersSet = {
    ? owner: Owner

    ~HTTPParameters

    ; qlog-specific
    ; indicates whether this implementation waits for a SETTINGS
    ; frame before processing requests
    ? waits_for_settings: bool
}

HTTPParameters = {
    ? max_header_list_size: uint64
    ? max_table_capacity: uint64
    ? blocked_streams_count: uint64

    ; additional settings for grease and extensions
    * text => uint64
}
```

Figure 2: HTTPParametersSet definition

Note: enabling server push is not explicitly done in HTTP/3 by use of a setting or parameter. Instead, it is communicated by use of the MAX_PUSH_ID frame, which should be logged using the frame_created and frame_parsed events below.

Additionally, this event can contain any number of unspecified fields. This is to reflect setting of for example unknown (greased) settings or parameters of (proprietary) extensions.

4.2. parameters_restored

Importance: Base

When using QUIC 0-RTT, HTTP/3 clients are expected to remember and reuse the server's SETTINGs from the previous connection. This event is used to indicate which HTTP/3 settings were restored and to which values when utilizing 0-RTT.

Definition:

```
HTTPParametersRestored = {

    ~HTTPParameters

}
```

Figure 3: HTTPParametersRestored definition

Note that, like for parameters_set above, this event can contain any number of unspecified fields to allow for additional and custom settings.

4.3. stream_type_set

Importance: Base

Emitted when a stream's type becomes known. This is typically when a stream is opened and the stream's type indicator is sent or received.

Note: most of this information can also be inferred by looking at a stream's id, since id's are strictly partitioned at the QUIC level. Even so, this event has a "Base" importance because it helps a lot in debugging to have this information clearly spelled out.

Definition:

```
HTTPStreamTypeSet = {
    ? owner: Owner
    stream_id: uint64

    stream_type: HTTPStreamType

    ; only when stream_type === "unknown"
    ? raw_stream_type: uint64

    ; only when stream_type === "push"
    ? associated_push_id: uint64
}

HTTPStreamType =  "request" /
                  "control" /
                  "push" /
                  "reserved" /
                  "unknown" /
                  "qpack_encode" /
                  "qpack_decode"
```

Figure 4: HTTPStreamTypeSet definition

4.4. frame_created

Importance: Core

HTTP equivalent to the packet_sent event. This event is emitted when the HTTP/3 framing actually happens. Note: this is not necessarily the same as when the HTTP/3 data is passed on to the QUIC layer. For that, see the "data_moved" event in [QLOG-QUIC].

Definition:

```
HTTPFrameCreated = {
    stream_id: uint64
    ? length: uint64
    frame: $HTTPFrame
    ? raw: RawInfo
}
```

Figure 5: HTTPFrameCreated definition

Note: in HTTP/3, DATA frames can have arbitrarily large lengths to reduce frame header overhead. As such, DATA frames can span many QUIC packets and can be created in a streaming fashion. In this case, the frame_created event is emitted once for the frame header, and further streamed data is indicated using the data_moved event.

4.5. frame_parsed

Importance: Core

HTTP equivalent to the packet_received event. This event is emitted when we actually parse the HTTP/3 frame. Note: this is not necessarily the same as when the HTTP/3 data is actually received on the QUIC layer. For that, see the "data_moved" event in [QLOG-QUIC].

Definition:

```
HTTPFrameParsed = {
    stream_id: uint64
    ? length: uint64
    frame: $HTTPFrame
    ? raw: RawInfo
}
```

Figure 6: HTTPFrameParsed definition

Note: in HTTP/3, DATA frames can have arbitrarily large lengths to reduce frame header overhead. As such, DATA frames can span many QUIC packets and can be processed in a streaming fashion. In this case, the frame_parsed event is emitted once for the frame header, and further streamed data is indicated using the data_moved event.

4.6. push_resolved

Importance: Extra

This event is emitted when a pushed resource is successfully claimed (used) or, conversely, abandoned (rejected) by the application on top of HTTP/3 (e.g., the web browser). This event is added to help debug problems with unexpected PUSH behaviour, which is commonplace with HTTP/2.

Definition:

```
HTTPPushResolved = {
    ? push_id: uint64

    ; in case this is logged from a place that does not have access
    ; to the push_id
    ? stream_id: uint64

    decision: HTTPPushDecision
}

HTTPPushDecision = "claimed" / "abandoned"
```

Figure 7: HTTPPushResolved definition

5. HTTP/3 Data Field Definitions

The following data field definitions can be used in HTTP/3 events.

5.1. Owner

```
Owner = "local" / "remote"
```

Figure 8: Owner definition

5.2. HTTPFrame

The generic $HTTPFrame is defined here as a CDDL extension point (a "socket" or "plug"). It can be extended to support additional HTTP/3 frame types.

```
; The HTTPFrame is any key-value map (e.g., JSON object)
$HTTPFrame /= {
    * text => any
}
```

Figure 9: HTTPFrame plug definition

The HTTP/3 frame types defined in this document are as follows:

```
HTTPBaseFrames =  HTTPDataFrame /
             HTTPHeadersFrame /
             HTTPCancelPushFrame /
             HTTPSettingsFrame /
             HTTPPushPromiseFrame /
             HTTPGoawayFrame /
             HTTPMaxPushIDFrame /
             HTTPReservedFrame /
             HTTPUnknownFrame

$HTTPFrame /= HTTPBaseFrames
```

Figure 10: HTTPBaseFrames definition

5.2.1. HTTPDataFrame

```
HTTPDataFrame = {
    frame_type: "data"
    ? raw: hexstring
}
```

Figure 11: HTTPDataFrame definition

5.2.2. HTTPHeadersFrame

This represents an *uncompressed*, plaintext HTTP Headers frame (e.g., no QPACK compression is applied).

For example:

```
headers: [
  {
    "name": ":path",
    "value": "/"
  },
  {
    "name": ":method",
    "value": "GET"
  },
  {
    "name": ":authority",
    "value": "127.0.0.1:4433"
  },
  {
    "name": ":scheme",
    "value": "https"
  }
]
```

Figure 12: HTTPHeadersFrame example

```
HTTPHeadersFrame = {
    frame_type: "headers"
    headers: [* HTTPField]
}
```

Figure 13: HTTPHeadersFrame definition

```
HTTPField = {
    name: text
    value: text
}
```

Figure 14: HTTPField definition

5.2.3. HTTPCancelPushFrame

```
HTTPCancelPushFrame = {
    frame_type: "cancel_push"
    push_id: uint64
}
```

Figure 15: HTTPCancelPushFrame definition

5.2.4. HTTPSettingsFrame

```
HTTPSettingsFrame = {
    frame_type: "settings"
    settings: [* HTTPSetting]
}

HTTPSetting = {
    name: text
    value: uint64
}
```

Figure 16: HTTPSettingsFrame definition

5.2.5. HTTPPushPromiseFrame

```
HTTPPushPromiseFrame = {
    frame_type: "push_promise"
    push_id: uint64
    headers: [* HTTPField]
}
```

Figure 17: HTTPPushPromiseFrame definition

5.2.6. HTTPGoAwayFrame

```
HTTPGoawayFrame = {
    frame_type: "goaway"

    ; Either stream_id or push_id.
    ; This is implicit from the sender of the frame
    id: uint64
}
```

Figure 18: HTTPGoawayFrame definition

5.2.7. HTTPMaxPushIDFrame

```
HTTPMaxPushIDFrame = {
    frame_type: "max_push_id"
    push_id: uint64
}
```

Figure 19: HTTPMaxPushIDFrame definition

5.2.8. HTTPReservedFrame

```
HTTPReservedFrame = {
    frame_type: "reserved"

    ? length: uint64
}
```

Figure 20: HTTPReservedFrame definition

5.2.9. HTTPUnknownFrame

```
HTTPUnknownFrame = {
    frame_type: "unknown"
    raw_frame_type: uint64

    ? raw_length: uint32
    ? raw: hexstring
}
```

Figure 21: UnknownFrame definition

5.2.10. HTTPApplicationError

```
HTTPApplicationError =  "http_no_error" /
                        "http_general_protocol_error" /
                        "http_internal_error" /
                        "http_stream_creation_error" /
                        "http_closed_critical_stream" /
                        "http_frame_unexpected" /
                        "http_frame_error" /
                        "http_excessive_load" /
                        "http_id_error" /
                        "http_settings_error" /
                        "http_missing_settings" /
                        "http_request_rejected" /
                        "http_request_cancelled" /
                        "http_request_incomplete" /
                        "http_early_response" /
                        "http_connect_error" /
                        "http_version_fallback"
```

Figure 22: HTTPApplicationError definition

The HTTPApplicationError defines the general $ApplicationError definition in the qlog QUIC definition, see [QLOG-QUIC].

```
; ensure HTTP errors are properly validate in QUIC events as well
; e.g., QUIC's ConnectionClose Frame
$ApplicationError /= HTTPApplicationError
```

6. QPACK Events

QPACK events extend the $ProtocolEventBody extension point defined in [QLOG-MAIN].

```
QPACKEvents = QPACKStateUpdate / QPACKStreamStateUpdate /
              QPACKDynamicTableUpdate / QPACKHeadersEncoded /
              QPACKHeadersDecoded / QPACKInstructionCreated /
              QPACKInstructionParsed

$ProtocolEventBody /= QPACKEvents
```

Figure 23: QPACKEvents definition and ProtocolEventBody extension

QPACK events mainly serve as an aid to debug low-level QPACK issues.The higher-level, plaintext header values **SHOULD** (also) be logged in the http.frame_created and http.frame_parsed event data (instead).

Note: qpack does not have its own parameters_set event. This was merged with http.parameters_set for brevity, since qpack is a required extension for HTTP/3 anyway. Other HTTP/3 extensions **MAY** also log their SETTINGS fields in http.parameters_set or **MAY** define their own events.

6.1. state_updated

Importance: Base

This event is emitted when one or more of the internal QPACK variables changes value. Note that some variables have two variations (one set locally, one requested by the remote peer). This is reflected in the "owner" field. As such, this field **MUST** be correct for all variables included a single event instance. If you need to log settings from two sides, you **MUST** emit two separate event instances.

Definition:

```
QPACKStateUpdate = {
    owner: Owner
    ? dynamic_table_capacity: uint64

    ; effective current size, sum of all the entries
    ? dynamic_table_size: uint64
    ? known_received_count: uint64
    ? current_insert_count: uint64
}
```

Figure 24: QPACKStateUpdate definition

6.2. stream_state_updated

Importance: Core

This event is emitted when a stream becomes blocked or unblocked by header decoding requests or QPACK instructions.

Note: This event is of "Core" importance, as it might have a large impact on HTTP/3's observed performance.

Definition:

```
QPACKStreamStateUpdate = {
    stream_id: uint64
    ; streams are assumed to start "unblocked"
    ; until they become "blocked"
    state: QPACKStreamState
}

QPACKStreamState = "blocked" / "unblocked"
```

Figure 25: QPACKStreamStateUpdate definition

6.3. dynamic_table_updated

Importance: Extra

This event is emitted when one or more entries are inserted or evicted from QPACK's dynamic table.

Definition:

```
QPACKDynamicTableUpdate = {
    ; local = the encoder's dynamic table
    ; remote = the decoder's dynamic table
    owner: Owner

    update_type: QPACKDynamicTableUpdateType
    entries: [+ QPACKDynamicTableEntry]
}

QPACKDynamicTableUpdateType = "inserted" / "evicted"

QPACKDynamicTableEntry = {
    index: uint64
    ? name: text / hexstring
    ? value: text / hexstring
}
```

Figure 26: QPACKDynamicTableUpdate definition

6.4. headers_encoded

Importance: Base

This event is emitted when an uncompressed header block is encoded successfully.

Note: this event has overlap with http.frame_created for the HeadersFrame type. When outputting both events, implementers **MAY** omit the "headers" field in this event.

Definition:

```
QPACKHeadersEncoded = {
    ? stream_id: uint64
    ? headers: [+ HTTPField]

    block_prefix: QPACKHeaderBlockPrefix
    header_block: [+ QPACKHeaderBlockRepresentation]

    ? length: uint
    ? raw: hexstring
}
```

Figure 27: QPACKHeadersEncoded definition

6.5. headers_decoded

Importance: Base

This event is emitted when a compressed header block is decoded successfully.

Note: this event has overlap with http.frame_parsed for the HeadersFrame type. When outputting both events, implementers **MAY** omit the "headers" field in this event.

Definition:

```
QPACKHeadersDecoded = {
    ? stream_id: uint64
    ? headers: [+ HTTPField]

    block_prefix: QPACKHeaderBlockPrefix
    header_block: [+ QPACKHeaderBlockRepresentation]

    ? length: uint32
    ? raw: hexstring
}
```

Figure 28: QPACKHeadersDecoded definition

6.6. instruction_created

Importance: Base

This event is emitted when a QPACK instruction (both decoder and encoder) is created and added to the encoder/decoder stream.

Definition:

```
QPACKInstructionCreated = {
    ; see definition in appendix
    instruction: QPACKInstruction
    ? length: uint32
    ? raw: hexstring
}
```

Figure 29: QPACKInstructionCreated definition

Note: encoder/decoder semantics and stream_id's are implicit in either the instruction types or can be logged via other events (e.g., http.stream_type_set)

6.7. instruction_parsed

Importance: Base

This event is emitted when a QPACK instruction (both decoder and encoder) is read from the encoder/decoder stream.

Definition:

```
QPACKInstructionParsed = {
    ; see QPACKInstruction definition in appendix
    instruction: QPACKInstruction

    ? length: uint32
    ? raw: hexstring
}
```

Figure 30: QPACKInstructionParsed definition

Note: encoder/decoder semantics and stream_id's are implicit in either the instruction types or can be logged via other events (e.g., http.stream_type_set)

7. QPACK Data Field Definitions

The following data field definitions can be used in QPACK events.

7.1. QPACKInstruction

Note: the instructions do not have explicit encoder/decoder types, since there is no overlap between the instructions of both types in neither name nor function.

```
QPACKInstruction =  SetDynamicTableCapacityInstruction /
                    InsertWithNameReferenceInstruction /
                    InsertWithoutNameReferenceInstruction /
                    DuplicateInstruction /
                    SectionAcknowledgementInstruction /
                    StreamCancellationInstruction /
                    InsertCountIncrementInstruction
```

Figure 31: QPACKInstruction definition

7.1.1. SetDynamicTableCapacityInstruction

```
SetDynamicTableCapacityInstruction = {
    instruction_type: "set_dynamic_table_capacity"
    capacity: uint32
}
```

Figure 32: SetDynamicTableCapacityInstruction definition

7.1.2. InsertWithNameReferenceInstruction

```
InsertWithNameReferenceInstruction = {
    instruction_type: "insert_with_name_reference"
    table_type: QPACKTableType
    name_index: uint32
    huffman_encoded_value: bool
    ? value_length: uint32
    ? value: text
}
```

Figure 33: InsertWithNameReferenceInstruction definition

7.1.3. InsertWithoutNameReferenceInstruction

```
InsertWithoutNameReferenceInstruction = {
    instruction_type: "insert_without_name_reference"
    huffman_encoded_name: bool
    ? name_length: uint32
    ? name: text
    huffman_encoded_value: bool
    ? value_length: uint32
    ? value: text
}
```

Figure 34: InsertWithoutNameReferenceInstruction definition

7.1.4. DuplicateInstruction

```
DuplicateInstruction = {
    instruction_type: "duplicate"
    index: uint32
}
```

Figure 35: DuplicateInstruction definition

7.1.5. SectionAcknowledgementInstruction

```
SectionAcknowledgementInstruction = {
    instruction_type: "section_acknowledgement"
    stream_id: uint64
}
```

Figure 36: SectionAcknowledgementInstruction definition

7.1.6. StreamCancellationInstruction

```
StreamCancellationInstruction = {
    instruction_type: "stream_cancellation"
    stream_id: uint64
}
```

Figure 37: StreamCancellationInstruction definition

7.1.7. InsertCountIncrementInstruction

```
InsertCountIncrementInstruction = {
    instruction_type: "insert_count_increment"
    increment: uint32
}
```

Figure 38: InsertCountIncrementInstruction definition

7.2. QPACKHeaderBlockRepresentation

```
QPACKHeaderBlockRepresentation =  IndexedHeaderField /
                                  LiteralHeaderFieldWithName /
                                  LiteralHeaderFieldWithoutName
```

Figure 39: QPACKHeaderBlockRepresentation definition

7.2.1. IndexedHeaderField

Note: also used for "indexed header field with post-base index"

```
IndexedHeaderField = {
    header_field_type: "indexed_header"

    ; MUST be "dynamic" if is_post_base is true
    table_type: QPACKTableType
    index: uint32

    ; to represent the "indexed header field with post-base index"
    ; header field type
    is_post_base: bool .default false
}
```

Figure 40: IndexedHeaderField definition

7.2.2. LiteralHeaderFieldWithName

Note: also used for "Literal header field with post-base name reference"

```
LiteralHeaderFieldWithName = {
    header_field_type: "literal_with_name"

    ; the 3rd "N" bit
    preserve_literal: bool

    ; MUST be "dynamic" if is_post_base is true
    table_type: QPACKTableType
    name_index: uint32
    huffman_encoded_value: bool
    ? value_length: uint32
    ? value: text

    ; to represent the "indexed header field with post-base index"
    ; header field type
    is_post_base: bool .default false
}
```

Figure 41: LiteralHeaderFieldWithName definition

7.2.3. LiteralHeaderFieldWithoutName

```
LiteralHeaderFieldWithoutName = {
    header_field_type: "literal_without_name"

    ; the 3rd "N" bit
    preserve_literal: bool
    huffman_encoded_name: bool
    ? name_length: uint32
    ? name: text

    huffman_encoded_value: bool
    ? value_length: uint32
    ? value: text
}
```

Figure 42: LiteralHeaderFieldWithoutName definition

7.3. QPACKHeaderBlockPrefix

```
QPACKHeaderBlockPrefix = {
    required_insert_count: uint32
    sign_bit: bool
    delta_base: uint32
}
```

Figure 43: QPACKHeaderBlockPrefix definition

7.4. QPACKTableType

```
QPACKTableType = "static" / "dynamic"
```

Figure 44: QPACKTableType definition

8. Security and Privacy Considerations

The security and privacy considerations discussed in [QLOG-MAIN] apply to this document as well.

9. IANA Considerations

TBD

10. Normative References

[CDDL] Birkholz, H., Vigano, C., and C. Bormann, "Concise Data Definition Language (CDDL): A Notational Convention to Express Concise Binary Object Representation (CBOR) and JSON Data Structures", RFC 8610, DOI 10.17487/RFC8610, June 2019, <https://www.rfc-editor.org/rfc/rfc8610>.

[HTTP/3]
Bishop, M., Ed., "HTTP/3", RFC 9114, DOI 10.17487/RFC9114, June 2022, <https://www.rfc-editor.org/rfc/rfc9114>.

[QLOG-MAIN] Marx, R., Niccolini, L., and M. Seemann, "Main logging schema for qlog", Work in Progress, Internet-Draft, draft-ietf-quic-qlog-main-schema-03, 31 August 2022, <https://datatracker.ietf.org/doc/html/draft-ietf-quic-qlog-main-schema-03>.

[QLOG-QUIC] Marx, R., Niccolini, L., and M. Seemann, "QUIC event definitions for qlog", Work in Progress, Internet-Draft, draft-ietf-quic-qlog-quic-events-02, 31 August 2022, <https://datatracker.ietf.org/doc/html/draft-ietf-quic-qlog-quic-events-02>.

[QPACK] Krasic, C., Bishop, M., and A. Frindell, Ed., "QPACK: Field Compression for HTTP/3", RFC 9204, DOI 10.17487/RFC9204, June 2022, <https://www.rfc-editor.org/rfc/rfc9204>.

[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/rfc/rfc2119>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/rfc/rfc8174>.

Appendix A. Change Log

A.1. Since draft-ietf-quic-qlog-h3-events-02:

* Renamed HTTPStreamType data to request (#222)

* Added HTTPStreamType value unknown (#227)

* Added HTTPUnknownFrame (#224)

* Replaced old and new fields with stream_type in HTTPStreamTypeSet (#240)

* Changed HTTPFrame to a CDDL plug type (#257)

* Moved data definitions out of the appendix into separate sections

* Added overview Table of Contents

A.2. Since draft-ietf-quic-qlog-h3-events-01:

* No changes - new draft to prevent expiration

A.3. Since draft-ietf-quic-qlog-h3-events-00:

* Change the data definition language from TypeScript to CDDL (#143)

A.4. Since draft-marx-qlog-event-definitions-quic-h3-02:

* These changes were done in preparation of the adoption of the drafts by the QUIC working group (#137)

* Split QUIC and HTTP/3 events into two separate documents

* Moved RawInfo, Importance, Generic events and Simulation events to the main schema document.

A.5. Since draft-marx-qlog-event-definitions-quic-h3-01:

Major changes:

* Moved data_moved from http to transport. Also made the "from" and "to" fields flexible strings instead of an enum (#111,#65)

* Moved packet_type fields to PacketHeader. Moved packet_size field out of PacketHeader to RawInfo:length (#40)

* Made events that need to log packet_type and packet_number use a header field instead of logging these fields individually

* Added support for logging retry, stateless reset and initial tokens (#94,#86,#117)

* Moved separate general event categories into a single category "generic" (#47)

* Added "transport:connection_closed" event (#43,#85,#78,#49)

* Added version_information and alpn_information events (#85,#75,#28)

* Added parameters_restored events to help clarify 0-RTT behaviour (#88)

Smaller changes:

* Merged loss_timer events into one loss_timer_updated event

* Field data types are now strongly defined (#10,#39,#36,#115)

* Renamed qpack instruction_received and instruction_sent to instruction_created and instruction_parsed (#114)

* Updated qpack:dynamic_table_updated.update_type. It now has the value "inserted" instead of "added" (#113)

* Updated qpack:dynamic_table_updated. It now has an "owner" field to differentiate encoder vs decoder state (#112)

* Removed push_allowed from http:parameters_set (#110)

* Removed explicit trigger field indications from events, since this was moved to be a generic property of the "data" field (#80)

* Updated transport:connection_id_updated to be more in line with other similar events. Also dropped importance from Core to Base (#45)

* Added length property to PaddingFrame (#34)

* Added packet_number field to transport:frames_processed (#74)

* Added a way to generically log packet header flags (first 8 bits) to PacketHeader

* Added additional guidance on which events to log in which situations (#53)

* Added "simulation:scenario" event to help indicate simulation details

* Added "packets_acked" event (#107)

* Added "datagram_ids" to the datagram_X and packet_X events to allow tracking of coalesced QUIC packets (#91)

* Extended connection_state_updated with more fine-grained states (#49)

A.6. Since draft-marx-qlog-event-definitions-quic-h3-00:

* Event and category names are now all lowercase

* Added many new events and their definitions

* "type" fields have been made more specific (especially important for PacketType fields, which are now called packet_type instead of type)

* Events are given an importance indicator (issue #22)

*Event names are more consistent and use past tense (issue #21)

*Triggers have been redefined as properties of the "data" field and updated for most events (issue #23)

Acknowledgements

Much of the initial work by Robin Marx was done at the Hasselt and KU Leuven Universities.

Thanks to Jana Iyengar, Brian Trammell, Dmitri Tikhonov, Stephen Petrides, Jari Arkko, Marcus Ihlar, Victor Vasiliev, Mirja Kuehlewind, Jeremy Laine, Kazu Yamamoto, and Christian Huitema for their feedback and suggestions.

Authors' Addresses

Robin Marx (editor)
Akamai

Email: rmarx@akamai.com

Luca Niccolini (editor)
Meta

Email: lniccolini@meta.com

Marten Seemann (editor)
Protocol Labs

Email: marten@protocol.ai

Lucas Pardue (editor)
Cloudflare

Email: lucaspardue.24.7@gmail.com