QUIC R. Marx
Internet-Draft Hasselt University
Intended status: Standards Track October 14, 2019
Expires: April 16, 2020

QUIC and HTTP/3 event definitions for qlog
draft-marx-qlog-event-definitions-quic-h3-01

Abstract

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

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 April 16, 2020.

Table of Contents

1. Introduction

This document describes the values of the qlog "category", "event" and "data" fields and their semantics for the QUIC and HTTP/3 protocols. This document is based on draft-23 of the QUIC and HTTP/3 I-Ds QUIC-TRANSPORT [QUIC-HTTP].

Feedback and discussion welcome at https://github.com/quiclog/internet-drafts [1]. 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/ [2].

1.1. Notational Conventions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [RFC2119].

This document uses the "TypeScript" language [3] to describe its schema in. We use TypeScript because it is less verbose than JSON-schema and almost as expressive. It also makes it easier to include these definitions directly into a web-based tool. TypeScript type definitions for this document are available at https://github.com/quiclog/qlog/tree/master/TypeScript. The main conventions a reader should be aware of are:

- obj? : this object is optional
- type1 | type2 : a union of these two types (object can be either type1 OR type2)
- obj:type : this object has this concrete type
- obj[] : this object is an array (which can contain any type of object)
- obj:Array<type> : this object is an array of this type
- number : identifies either an integer, float or double in TypeScript. In this document, number always means an integer.
- Unless explicity defined, the value of an enum entry is the string version of its name (e.g., initial = "initial")
- Many numerical fields have type "string" instead of "number". This is because many JSON implementations only support integers up to 2^53-1 (MAX_INTEGER for JavaScript without BigInt support), which is less than QUIC's VLIE types (2^62-1). Each field that can potentially have a value larger than 2^53-1 is thus a string, where a number would be semantically more correct. Unless mentioned otherwise (e.g., for connection IDs), numerical fields

that are logged as strings (e.g., packet numbers) MUST be logged in decimal (base-10) format. TODO: see issue 10

1.2. Importance

Not all the listed events are of equal importance to achieve good debuggability. As such, each event has an "importance indicator" with one of three values, in decreasing order of importance and exptected usage:

- Core
- Base
- Extra

The "Core" events are the events that SHOULD be present in all qlog files. These are mostly tied to basic packet and frame parsing and creation, as well as listing basic internal metrics. Tool implementers SHOULD expect and add support for these events, though SHOULD NOT expect all Core events to be present in each qlog trace.

The "Base" events add additional debugging options and CAN be present in qlog files. Most of these can be implicitly inferred from data in Core events (if those contain all their properties), but for many it is better to log the events explicitly as well, making it clearer how the implementation behaves. These events are for example tied to passing data around in buffers, to how internal state machines change and help show when decisions are actually made based on received data. Tool implementers SHOULD at least add support for showing the contents of these events, if they do not handle them explicitly.

The "Extra" events are considered mostly useful for low-level debugging of the implementation, rather than the protocol. They allow more fine-grained tracking of internal behaviour. As such, they CAN be present in qlog files and tool implementers CAN add support for these, but they are not required to.

Note that in some cases, implementers might not want to log frame-level details in the "Core" events due to performance considerations. In this case, they SHOULD use (a subset of) relevant "Base" events instead to ensure usability of the qlog output. As an example, implementations that do not log "packet_received" events and thus also not which (if any) ACK frames the packet contain, SHOULD log packets_acknowledged events instead.

2. Overview

This document describes the values of the qlog "category", "event" and "data" fields and their semantics for the QUIC and HTTP/3 protocols. The definitions included in this file are assumed to be used in qlog's "trace" containers, where the trace's "protocol_type" field MUST be set to "QUIC_HTTP3".

Many of the events map directly to concepts seen in the QUIC and HTTP/3 documents, while others act as aggregating events that combine data from several possible protocol behaviours or code paths into one, to reduce the amount of different event definitions. Limiting the amount of different events is one of the main design goals for this document. As such, many events that can be directly inferred from data on the wire (e.g., flow control limit changes) if the implementation is bug-free, are not explicitly defined as stand-alone events.

Similarly, we prevent logging duplicate data as much as possible. As such, especially packet header value updates are split out into separate events (e.g., spin_bit_updated, connection_id_updated), as they are expected to change sparingly.

This document assumes the usage of the encompassing main qlog schema defined in [QLOG-MAIN]. Each subsection below defines a separate category (e.g., connectivity, transport, http) and each subsubsection is an event type.

For each event type, its importance and data definition is laid out, often accompanied by possible values for the optional "trigger" field. For the definition and semantics of "trigger", see the main scheme document.

Most of the complex datastructures, enums and re-usable definitions are grouped together on the bottom of the document for clarity.

2.1. Custom fields

Note that implementers are free to define new category and event types, as well as values for the "trigger" property within the "data" field, as they see fit. They SHOULD NOT however expect non-specialized tools to recognize or visualize this custom data. However, tools SHOULD make an effort to visualize even unknown data if possible in the specific tool's context.

3. Events not belonging to a single connection

For several types of events, it is sometimes impossible to tie them to a specific conceptual QUIC connection (e.g., a packet_dropped event triggered because the packet has an unknown connection_id in the header). Since a qlog events in a trace are typically associated with a single connection (see the discussions on group_id in draft-marx-quic-logging-main-schema-latest), it is unclear how to log these events.

Ideally, implementers SHOULD create a separate "endpoint-level" trace or at least group_id, not associated with a specific connection (e.g., group_id = "server" | "client"), and log all of these events on that trace. However, this is not always practical, depending on the implementation. Because the semantics of these events are well-defined in the protocols and because they are difficult to mis-interpret as belonging to a connection, implementers MAY log events not belonging to a particular connection in any other trace, even those strongly associated with a single connection.

Note that this can make it difficult to match logs from different vantage points with each other. For example, from the client side, it is easy to log connections with version negotiation or stateless retry in the same trace, while on the server they would most likely be logged in separate traces.

4. QUIC and HTTP/3 fields

This document re-uses all the fields defined in the main qlog schema (e.g.,, category, event, data, group_id, protocol_type, the time-related fields, etc.).

The value of the protocol_type field MUST be "QUIC_HTTP3".

As the group_id field can contain any grouping identifier, this document defines an additional similar field, named ODCID (for Original Destination Connection ID), since the ODCID is the lowest common denominator to be able to link packets to a connection. Typically though, the group_id and ODCID fields will contain the same value (or the ODCID field is omitted).

5. QUIC event definitions

5.1. connectivity

5.1.1. server_listening

Importance: Extra

Emitted when the server starts accepting connections.

Data:

```
{
    ip_v4?: string,
    ip_v6?: string,
    port_v4: number,
    port_v6: number,

    quic_versions?: Array<string>,
    alpn_values?: Array<string>,

    stateless_reset_required?:boolean // server will always respond
with stateless_reset for incoming initials
}
```

5.1.2. connection_started

Importance: Base

Used for both attempting (client-perspective) and accepting (server-perspective) new connections. Note that this event has overlap with connection_state_updated and this is a separate event mainly because of all the additional data that should be logged.

Data:

```
{
    ip_version: string,
    src_ip: string,
    dst_ip: string,

    protocol?: string, // (default "QUIC")
    src_port: number,
    dst_port: number,

    quic_version?: string,
    src_cid?: string,
    dst_cid?: string
}
```

5.1.3. connection_id_updated

Importance: Core

This is viewed from the perspective of the one applying the new id. As such, if we receive a new connection id from our peer, we will see the dst_ fields are set. If we update our own connection id (e.g., NEW_CONNECTION_ID frame), we log the src_ fields.

Data:

```
{
    src_old?: string,
    src_new?: string,

    dst_old?: string,
    dst_new?: string
}
```

5.1.4. spin_bit_updated

Importance: Base

To be emitted when the spin bit changes value. It SHOULD NOT be emitted if the spin bit is set without changing its value.

Data:

```
{
    state: boolean
}
```

5.1.5. connection_retried

TODO

5.1.6. connection_state_updated

Importance: Base

Data:

```
{
    old?:ConnectionState,
    new:ConnectionState
}

enum ConnectionState {
    attempted, // client initial sent
    reset, // stateless reset sent
    handshake, // handshake in progress
    active, // handshake successful, data exchange
    keepalive, // no data for a longer period
    draining, // CONNECTION_CLOSE sent
    closed // connection actually fully closed, memory freed
}
```

Note: connection_state_changed with a new state of "attempted" is the same conceptual event as the connection_started event above from the client's perspective.

Triggers:

- "error" // when closing because of an unexpected event
- "clean" // when closing normally
- "application" // e.g., HTTP/3's GOAWAY frame

5.1.7. MIGRATION-related events

e.g., path_updated

TODO: read up on the draft how migration works and whether to best fit this here or in TRANSPORT TODO: integrate https://tools.ietf.org/html/draft-deconinck-quic-multipath-02

For now, infer from other connectivity events and path_challenge/ path_response frames

5.2. security

5.2.1. key_updated

Importance: Base

Note: secret_updated would be more correct, but in the draft it's called KEY_UPDATE, so stick with that for consistency

Data:

```
{
    key_type:KeyType,
    old?:string,
    new:string,
    generation?:number, // needed for 1RTT key updates

    trigger?: string
}
```

Triggers:

- "tls" // (e.g., initial, handshake and 0-RTT keys are generated by TLS)
- "remote_update"
- "local_update"

5.2.2. key_retired

Importance: Base

Data:

```
{
    key_type:KeyType,
    key?:string,
    generation?:number, // needed for 1RTT key updates

    trigger?: string
}
```

Triggers:

- "tls" // (e.g., initial, handshake and 0-RTT keys are dropped implicitly)
- "remote_update"
- "local_update"

5.3. transport

5.3.1. parameters_set

Importance: Core

This event groups settings from many different sources (transport parameters, version negotiation, ALPN selection, TLS ciphers, etc.) into a single event. This is done to minimize the amount of events and to decouple conceptual setting impacts from their underlying mechanism for easier high-level reasoning.

All these settings are typically set once and never change. However, they are typically set at different times during the connection, so there will typically 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.

Data:

```
{
    owner?:"local" | "remote", // can be left for bidirectionally
negotiated parameters, e.g. ALPN

    resumption_allowed?:boolean, // valid session ticket was received
    early_data_enabled?:boolean, // early data extension was enabled on
the TLS layer
    alpn?:string,
    version?:string, // hex (e.g., 0x)
    tls_cipher?:string, // (e.g., AES_128_GCM_SHA256)

    // transport parameters from the TLS layer:
    original_connection_id?:string, // hex
    stateless_reset_token?:string, // hex
    disable_active_migration?:boolean,

    idle_timeout?:number,
    max_packet_size?:number,
    ack_delay_exponent?:number,
    max_ack_delay?:number,
    active_connection_id_limit?:number,

    initial_max_data?:string,
    initial_max_stream_data_bidi_local?:string,
    initial_max_stream_data_bidi_remote?:string,
    initial_max_stream_data_uni?:string,
    initial_max_streams_bidi?:string,
    initial_max_streams_uni?:string,

    preferred_address?:PreferredAddress
}

interface PreferredAddress {
    ip_v4:string,
    ip_v6:string,

    port_v4:number,
    port_v6:number,

    connection_id:string, // hex
    stateless_reset_token:string // hex
}
```

Additionally, this event can contain any number of unspecified fields. This is to reflect setting of for example unknown (greased) transport parameters or employed (proprietary) extensions. In this case, the field name should be the hex-encoded value of the parameter name or identifier.

5.3.2. packet_sent

Importance: Core

Data:

```
{
    packet_type:PacketType,
    header:PacketHeader,
    frames?:Array<QuicFrame>, // see appendix for the definitions

    is_coalesced?:boolean,

    raw_encrypted?:string, // for debugging purposes
    raw_decrypted?:string  // for debugging purposes,

    trigger?: string
}
```

Note: We do not explicitly log the encryption_level or packet_number_space: the packet_type specifies this by inference (assuming correct implementation)

Triggers:

- "retransmit_reordered" // draft-23 5.1.1
- "retransmit_timeout" // draft-23 5.1.2
- "pto_probe" // draft-23 5.3.1
- "retransmit_crypto" // draft-19 6.2
- "cc_bandwidth_probe" // needed for some CCs to figure out bandwidth allocations when there are no normal sends

5.3.3. packet_received

Importance: Core

Data:

```
{
    packet_type:PacketType,
    header:PacketHeader,
    frames?:Array<QuicFrame>, // see appendix for the definitions

    is_coalesced?:boolean,

    raw_encrypted?:string, // for debugging purposes
    raw_decrypted?:string  // for debugging purposes,

    trigger?: string
}
```

Note: We do not explicitly log the encryption_level or packet_number_space: the packet_type specifies this by inference (assuming correct implementation)

Triggers:

- "keys_available" // if packet was buffered because it couldn't be decrypted before

5.3.4. packet_dropped

Importance: Base

This event indicates a QUIC-level packet was dropped after partial or no parsing.

For this event, the "trigger" property SHOULD be set to one of the values below, as this helps tremendously in debugging.

Data:

```
{
    packet_type?:PacketType,
    packet_size?:number,
    raw?:string, // hex encoded,

    trigger?: string
}
```

Triggers:

- "key_unavailable"

- "unknown_connection_id"

- "header_decrypt_error"
- "payload_decrypt_error"
- "protocol_violation"
- "dos_prevention"
- "unsupported_version"

Note: sometimes packets are dropped before they can be associated with a particular connection (e.g., in case of "unsupported_version"). This situation is discussed in Section 3.

5.3.5. packet_buffered

Importance: Base

This event is emitted when a packet is buffered because it cannot be processed yet. Typically, this is because the packet cannot be parsed yet, and thus we only log the full packet contents when it was parsed in a packet_received event.

Data:

```
{
    packet_type:PacketType,
    packet_number?:string,

    trigger?: string
}
```

Triggers:

- "backpressure" // indicates the parser cannot keep up, temporarily buffers packet for later processing
- "keys_unavailable" // if packet cannot be decrypted because the proper keys were not yet available

5.3.6. datagrams_sent

Importance: Extra

When we pass one or more UDP-level datagrams to the socket. This is useful for determining how QUIC packet buffers are drained to the OS.

Data:

```
{
    count?:number, // to support passing multiple at once
    byte_length?:number
}
```

5.3.7. datagrams_received

Importance: Extra

When we receive one or more UDP-level datagrams from the socket. This is useful for determining how datagrams are passed to the user space stack from the OS.

Data:

```
{
    count?:number, // to support passing multiple at once
    byte_length?:number
}
```

5.3.8. datagram_dropped

Importance: Extra

When we drop a UDP-level datagram. This is typically if it does not contain a valid QUIC packet (in that case, use packet_dropped instead).

Data:

```
{
    byte_length?:number
}
```

5.3.9. stream_state_updated

Importance: Base

This event is emitted whenever the internal state of a QUIC stream is updated, as described in QUIC transport draft-23 section 3. Most of this can be inferred from several types of frames going over the wire, but it's much easier to have explicit signals for these state changes.

Data:

```
{
    stream_id:string,
    stream_type?:"unidirectional"|"bidirectional", // mainly useful
when opening the stream

    old?:StreamState,
    new:StreamState,

    stream_side?:"sending"|"receiving"
}

enum StreamState {
    // bidirectional stream states, draft-23 3.4.
    idle,
    open,
    half_closed_local,
    half_closed_remote,
    closed,

    // sending-side stream states, draft-23 3.1.
    ready,
    send,
    data_sent,
    reset_sent,
    reset_received,

    // receive-side stream states, draft-23 3.2.
    receive,
    size_known,
    data_read,
    reset_read,

    // both-side states
    data_received,

    // qlog-defined
    destroyed // memory actually freed
}
```

Note: QUIC implementations SHOULD mainly log the simplified bidirectional (HTTP/2-alike) stream states (e.g., idle, open, closed) instead of the more finegrained stream states (e.g., data_sent, reset_received). These latter ones are mainly for more in-depth debugging. Tools SHOULD be able to deal with both types equally.

5.3.10. frames_processed

Importance: Extra

This event's main goal is to prevent a large proliferation of specific purpose events (e.g., packets_acknowledged, flow_control_updated, stream_data_received). We want to give implementations the opportunity to (selectively) log this type of signal without having to log packet-level details (e.g., in packet_received). Since for almost all cases, the effects of applying a frame to the internal state of an implementation can be inferred from that frame's contents, we aggregate these events in this single "frames_processed" event.

Note: This event can be used to signal internal state change not resulting directly from the actual "parsing" of a frame (e.g., the frame could have been parsed, data put into a buffer, then later processed, then logged with this event).

Note: Implementations logging "packet_received" and which include all of the packet's constituent frames therein, are not expected to emit this "frames_processed" event (contrary to the HTTP-level "frames_parsed" event). Rather, implementations not wishing to log full packets or that wish to explicitly convey extra information about when frames are processed (if not directly tied to their reception) can use this event.

Note: for some events, this approach will lose some information (e.g., for which encryption level are packets being acknowledged?). If this information is important, please use the packet_received event instead.

Data:

```
{
    frames:Array<QuicFrame>, // see appendix for the definitions
}
```

5.4. recovery

Note: most of the events in this category are kept generic to support different recovery approaches and various congestion control algorithms. Tool creators SHOULD make an effort to support and visualize even unknown data in these events (e.g., plot unknown congestion states by name on a timeline visualization).

5.4.1. parameters_set

Importance: Base

This event groups initial parameters from both loss detection and congestion control into a single event. All these settings are typically set once and never change. Implementation that do, for some reason, change these parameters during execution, MAY emit the parameters_set event twice.

Data:

```
{
    // Loss detection, see recovery draft-23, Appendix A.2
    reordering_threshold?:number, // in amount of packets
    time_threshold?:number, // as RTT multiplier
    timer_granularity?:number, // in ms or us, depending on the
overarching qlog's configuration
    initial_rtt?:number, // in ms or us, depending on the overarching
qlog's configuration

    // congestion control, Appendix B.1.
    max_datagram_size?:number, // in bytes // Note: this could be
updated after pmtud
    initial_congestion_window?:number, // in bytes
    minimum_congestion_window?:number, // in bytes // Note: this could
change when max_datagram_size changes
    loss_reduction_factor?:number,
    persistent_congestion_threshold?:number // as PTO multiplier
}
```

Additionally, this event can contain any number of unspecified fields
to support different recovery approaches.

5.4.2. metrics_updated

Importance: Core

This event is emitted when one or more of the observable recovery metrics changes value. This event SHOULD group all possible metric updates that happen at or around the same time in a single event (e.g., if min_rtt and smoothed_rtt change at the same time, they should be bundled in a single metrics_updated entry, rather than split out into two). Consequently, a metrics_updated event is only guaranteed to contain at least one of the listed metrics.

Data:

```
{
    // Loss detection, see recovery draft-23, Appendix A.3
    min_rtt?:number, // in ms or us, depending on the overarching
qlog's configuration
    smoothed_rtt?:number, // in ms or us, depending on the overarching
qlog's configuration
    latest_rtt?:number, // in ms or us, depending on the overarching
qlog's configuration
    rtt_variance?:number, // in ms or us, depending on the overarching
qlog's configuration

    max_ack_delay?:number, // in ms or us, depending on the overarching
qlog's configuration
    pto_count?:number,

    // Congestion control, Appendix B.2.
    congestion_window?:number, // in bytes
    bytes_in_flight?:number,

    ssthresh?:number, // in bytes

    // qlog defined
    packets_in_flight?:number, // sum of all packet number spaces
    in_recovery?:boolean, // high-level signal. For more granularity,
see congestion_state_updated

    pacing_rate?:number // in bps
}
```

Note: to make logging easier, implementations MAY log values even if they are the same as previously reported values (e.g., two subsequent METRIC_UPDATE entries can both report the exact same value for min_rtt). However, applications SHOULD try to log only actual updates to values.

Additionally, this event can contain any number of unspecified fields to support different recovery approaches.

5.4.3. congestion_state_updated

Importance: Base

This event signifies when the congestion controller enters a significant new state and changes its behaviour. This event's definition is kept generic to support different Congestion Control algorithms. For example, for the algorithm defined in the Recovery draft ("enhanced" New Reno), the following states are defined:

- slow_start

- congestion_avoidance
- application_limited

- recovery

The trigger SHOULD be logged if there are multiple ways in which a state change can occur but MAY be omitted if a given state can only be due to a single event occuring (e.g., slow start is exited only when ssthresh is exceeded).

Some triggers for ("enhanced" New Reno):

- persistent_congestion

- ECN

Data:

```
{
    old?:string,
    new:string,

    trigger?:string
}
```

5.4.4. loss_timer_set

Importance: Extra

This event is emitted when the single recovery loss timer is set.

Data:

```
{
    timer_type?:"ack"|"pto", // called "mode" in draft-23 A.9.
    timeout?:number
}
```

TODO: how about CC algo's that use multiple timers? How generic do these events need to be? Just support QUIC-style recovery from the spec or broader?

Triggers:

TODO

5.4.5. loss_timer_expired

Importance: Extra

This event is emitted when the single recovery loss timer fires.

Data:

```
{
    timer_type?:"ack"|"pto", // called "mode" in draft-23 A.9.
}
```

5.4.6. packet_lost

Importance: Core

This event is emitted when a packet is deemed lost by loss detection.
Use the "trigger" field to indicate the loss detection method used for this decision.

Data:

```
{
    packet_type:PacketType,
    packet_number:string,

    // not all implementations will keep track of full packets, so these are optional
    header?:PacketHeader,
    frames?:Array<QuicFrame>, // see appendix for the definitions,

    trigger?: string
}
```

Triggers:

- "reordering_threshold",
- "time_threshold"
- "pto_expired" // draft-23 section 5.3.1, MAY

5.4.7. marked_for_retransmit

Importance: Extra

This event indicates which data was marked for retransmit upon detecing a packet loss (see packet_lost). Similar to our reasoning for the "frames_processed" event, in order to keep the amount of different events low, we group this signal for all types of retransmittable data in a single event based on existing QUIC frame definitions.

Implementations retransmitting full packets or frames directly can just log the consituent frames of the lost packet here (or do away

with this event and use the contents of the packet_lost event instead). Conversely, implementations that have more complex logic (e.g., marking ranges in a stream's data buffer as in-flight), or that do not track sent frames in full (e.g., only stream offset + length), can translate their internal behaviour into the appropriate frame instance here even if that frame was never or will never be put on the wire.

Note: much of this data can be inferred if implementations log packet_sent events (e.g., looking at overlapping stream data offsets and length, one can determine when data was retransmitted).

Data:

```
{
    frames:Array<QuicFrame>, // see appendix for the definitions
}
```

6. HTTP/3 event definitions

6.1. http

Note: like all category values, the "http" category is written in lowercase.

6.1.1. 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.

Data:

```
{
    owner?:"local" | "remote",

    max_header_list_size?:number, // from SETTINGS_MAX_HEADER_LIST_SIZE
    max_table_capacity?:number, // from
SETTINGS_QPACK_MAX_TABLE_CAPACITY
    blocked_streams_count?:number, // from
SETTINGS_QPACK_BLOCKED_STREAMS

    push_allowed?:boolean, // received a MAX_PUSH_ID frame with non-
zero value

    // qlog-defined
    waits_for_settings?:boolean // indicates whether this
implementation waits for a SETTINGS frame before processing requests
}
```

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. In this case, the field name should be the hex-encoded value of the setting identifier.

6.1.2. 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.

Data:

```
{
    stream_id:string,

    owner?:"local"|"remote"

    old?:StreamType,
    new:StreamType,

    associated_push_id?:number // only when new == "push"
}

enum StreamType {
    data, // bidirectional request-response streams
    control,
    push,
    reserved,
    qpack_encode,
    qpack_decode
}
```

6.1.3. 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.

Data:

```
{
    stream_id:string,
    frame:HTTP3Frame // see appendix for the definitions,
    byte_length?:string,

    raw?:string // in hex
}
```

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.

6.1.4. 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.

Data:

```
{
    stream_id:string,
    frame:HTTP3Frame // see appendix for the definitions,
    byte_length?:string,

    raw?:string // in hex
}
```

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.

6.1.5. data_moved

Importance: Base

Used to indicate when data moves between the HTTP/3 and the transport layer (e.g., passing from H3 to QUIC stream buffers and vice versa) or between HTTP/3 and the actual user application on top (e.g., a browser engine). This helps make clear the flow of data, how long data remains in various buffers and the overheads introduced by HTTP/3's framing layer.

For example, when moving from application to http, the data will most likely be the raw request we wish to transmit. When then moving that request from http to transport, it will be compressed using QPACK and wrapped in an HTTP/3 HEADERS frame. Similarly, when receiving data from the transport layer, this will potentially include HTTP/3 headers, which are not passed on to the application layer. A final use case is making clear when only part of an HTTP/3 frame is received (e.g., only 1 or 2 bytes, while 3, 4 or more are needed to fully interpret an HTTP/3 frame).

Data:

```
{
    stream_id:string,
    offset?:string,
    length?:number,

    from?:"application"|"transport",
    to?:"application"|"transport",

    raw?:string // in hex
}
```

The "from" and "to" fields MUST NOT be set at the same time. The missing field is always implied to have the value "http".

6.1.6. 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.

```
{
    push_id?:number,
    stream_id?:string, // in case this is logged from a place that does not have access to the push_id

    decision:"claimed"|"abandoned"
}
```

6.2. qpack

Note: like all category values, the "qpack" category is written in lowercase.

The 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.

6.2.1. 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.

Data:

```
{
    owner?:"local" | "remote", // can be left for bidirectionally
negotiated parameters, e.g. ALPN

    dynamic_table_capacity?:number,
    dynamic_table_size?:number, // effective current size, sum of all
the entries

    known_received_count?:number,
    current_insert_count?:number
}
```

6.2.2. 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.

Data:

```
{
    stream_id:string,

    state:"blocked"|"unblocked" // streams are assumed to start
"unblocked" until they become "blocked"
}
```

6.2.3. dynamic_table_updated

Importance: Extra

This event is emitted when one or more entries are added or evicted

from QPACK's dynamic table.

```
Data:

{
    update_type:"added"|"evicted",

    entries:Array<DynamicTableEntry>
}

class DynamicTableEntry {
    index:number;
    name?:string;
    value?:string;
}
```

6.2.4. 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.

```
Data:

{
    stream_id?:string,

    headers?:Array<HTTPHeader>,

    block_prefix:QPackHeaderBlockPrefix,
    header_block:Array<QPackHeaderBlockRepresentation>,

    raw?:string, // in hex
}
```

6.2.5. 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.

```
Data:

{
    stream_id?:string,

    headers?:Array<HTTPHeader>,

    block_prefix:QPackHeaderBlockPrefix,
    header_block:Array<QPackHeaderBlockRepresentation>,

    raw?:string, // in hex
}
```

6.2.6. instruction_sent

Importance: Base

This event is emitted when a QPACK instruction (both decoder and encoder) is sent.

```
Data:

{
    instruction:QPackInstruction // see appendix for the definitions,
    byte_length?:string,

    raw?:string // in hex
}
```

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)

6.2.7. instruction_received

Importance: Base

This event is emitted when a QPACK instruction (both decoder and encoder) is received.

```
Data:

{
    instruction:QPackInstruction // see appendix for the definitions,
    byte_length?:string,

    raw?:string // in hex
}
```

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. General error, warning and debugging definitions

7.1. error

7.1.1. connection_error

Importance: Core

Logged when there is a connection error. Can be inferred from a CONNECTION_CLOSE frame, but one might refrain from sending a long string in that frame, while logging it here.

Data:

```
{
    code?:TransportError | CryptoError | number,
    description?:string
}
```

7.1.2. application_error

Importance: Core

Logged when there is an application error. Can be inferred from a CONNECTION_CLOSE frame, but one might refrain from sending a long string in that frame, while logging it here.

Data:

```
{
    code?:ApplicationError | number,
    description?:string
}
```

7.1.3. internal_error

Importance: Base

Used to log details of an internal error that might get translated into a more generic error on the wire (e.g., protocol_violation)

Data:

```
{
    code?:number,
    description?:string
}
```

7.2. warning

7.2.1. internal_warning

Importance: Base

Used to log details of an internal warning that might not get reflected on the wire.

Data:

```
{
    code?:number,
    description?:string
}
```

7.3. info

7.3.1. message

Importance: Extra

Used mainly for implementations that want to use qlog as their one and only logging format but still want to support unstructured string messages.

Data:

```
{
    message:string
}
```

7.4. debug

7.4.1. message

Importance: Extra

Used mainly for implementations that want to use qlog as their one and only logging format but still want to support unstructured string messages.

Data:

```
{
    message:string
}
```

7.5. verbose

7.5.1. message

Importance: Extra

Used mainly for implementations that want to use qlog as their one and only logging format but still want to support unstructured string messages.

Data:

```
{
    message:string
}
```

7.6. simulation

7.6.1. marker

Importance: Extra

Used for when running an implementation in a form of simulation setup where specific emulation conditions are triggered at set times (e.g., at 3 seconds in 2% packet loss is introduced, at 10s a NAT rebind is triggered). Marker events can be added to the logs and visualizations to show clearly when underlying conditions have been changed.

```
{
    marker_type:string,
    message?:string
}
```

8. Security Considerations

TBD

9. IANA Considerations

TBD

10. References

10.1. Normative References

[QLOG-MAIN]
Marx, R., Ed., "Main logging schema for qlog", draft-marx-qlog-main-schema-01 (work in progress), October 2019.

[QUIC-HTTP]
Bishop, M., Ed., "Hypertext Transfer Protocol Version 3 (HTTP/3)", draft-ietf-quic-http-23 (work in progress), September 2019.

[QUIC-TRANSPORT]
Iyengar, J., Ed. and M. Thomson, Ed., "QUIC: A UDP-Based Multiplexed and Secure Transport", draft-ietf-quic-transport-23 (work in progress), September 2019.

[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>.

10.2. URIs

[1] https://github.com/quiclog/internet-drafts

[2] https://github.com/quiclog/qlog/

[3] https://www.typescriptlang.org/

Appendix A. QUIC data field definitions

A.1. PacketType

```
enum PacketType {
    initial,
    handshake,
    zerortt = "0RTT",
    onertt = "1RTT",
    retry,
    version_negotiation,
    unknown
}
```

A.2. PacketHeader

```
class PacketHeader {
    packet_number: string;
    packet_size?: number;
    payload_length?: number;

    // only if present in the header
    // if correctly using NEW_CONNECTION_ID events,
    // dcid can be skipped for 1RTT packets
    version?: string;
    scil?: string;
    dcil?: string;
    scid?: string;
    dcid?: string;

    // Note: short vs long header is implicit through PacketType
}
```

A.3. KeyType

```
enum KeyType {
    server_initial_secret,
    client_initial_secret,

    server_handshake_secret,
    client_handshake_secret,

    server_0rtt_secret,
    client_0rtt_secret,

    server_1rtt_secret,
    client_1rtt_secret
}
```

A.4. QUIC Frames

```
type QuicFrame = PaddingFrame | PingFrame | AckFrame | ResetStreamFrame
| StopSendingFrame | CryptoFrame | NewTokenFrame | StreamFrame |
MaxDataFrame | MaxStreamDataFrame | MaxStreamsFrame | DataBlockedFrame
| StreamDataBlockedFrame | StreamsBlockedFrame | NewConnectionIDFrame |
RetireConnectionIDFrame | PathChallengeFrame | PathResponseFrame |
ConnectionCloseFrame | UnknownFrame;
```

A.4.1. PaddingFrame

```
class PaddingFrame{
    frame_type:string = "padding";
}
```

A.4.2. PingFrame

```
   class PingFrame{
       frame_type:string = "ping";
   }
```

A.4.3. AckFrame

```
class AckFrame{
    frame_type:string = "ack";

    ack_delay?:string; // in ms or us, depending on the overarching
qlog's configuration

    // first number is "from": lowest packet number in interval
    // second number is "to": up to and including // highest packet
number in interval
    // e.g., looks like [["1","2"],["4","5"]]
    acked_ranges?:Array<[string, string]|[string]>;

    ect1?:string;
    ect0?:string;
    ce?:string;
}
```

Note: the packet ranges in AckFrame.acked_ranges do not necessarily have to be ordered (e.g., [["5","9"],["1","4"]] is a valid value).

Note: the two numbers in the packet range can be the same (e.g., ["120","120"] means that packet with number 120 was ACKed). However, in that case, implementers SHOULD log ["120"] instead and tools MUST be able to deal with both notations.

A.4.4. ResetStreamFrame

```
   class ResetStreamFrame{
       frame_type:string = "reset_stream";

       stream_id:string;
       error_code:ApplicationError | number;
       final_size:string;
   }
```

A.4.5. StopSendingFrame

```
   class StopSendingFrame{
       frame_type:string = "stop_sending";

       stream_id:string;
       error_code:ApplicationError | number;
   }
```

A.4.6. CryptoFrame

```
   class CryptoFrame{
     frame_type:string = "crypto";

     offset:string;
     length:number;
   }
```

A.4.7. NewTokenFrame

```
   class NewTokenFrame{
     frame_type:string = "new_token";

     length:number;
     token:string;
   }
```

A.4.8. StreamFrame

```
class StreamFrame{
    frame_type:string = "stream";

    stream_id:string;

    // These two MUST always be set
    // If not present in the Frame type, log their default values
    offset:string;
    length:number;

    // this MAY be set any time, but MUST only be set if the value is
"true"
    // if absent, the value MUST be assumed to be "false"
    fin?:boolean;

    raw?:string;
}
```

A.4.9. MaxDataFrame

```
class MaxDataFrame{
  frame_type:string = "max_data";

  maximum:string;
}
```

A.4.10. MaxStreamDataFrame

```
class MaxStreamDataFrame{
  frame_type:string = "max_stream_data";

  stream_id:string;
  maximum:string;
}
```

A.4.11. MaxStreamsFrame

```
class MaxStreamsFrame{
  frame_type:string = "max_streams";

  stream_type:string = "bidirectional" | "unidirectional";
  maximum:string;
}
```

A.4.12. DataBlockedFrame

```
class DataBlockedFrame{
  frame_type:string = "data_blocked";

  limit:string;
}
```

A.4.13. StreamDataBlockedFrame

```
class StreamDataBlockedFrame{
  frame_type:string = "stream_data_blocked";

  stream_id:string;
  limit:string;
}
```

A.4.14. StreamsBlockedFrame

```
class StreamsBlockedFrame{
  frame_type:string = "streams_blocked";

  stream_type:string = "bidirectional" | "unidirectional";
  limit:string;
}
```

A.4.15. NewConnectionIDFrame

```
class NewConnectionIDFrame{
  frame_type:string = "new_connection_id";

  sequence_number:string;
  retire_prior_to:string;

  length:number;
  connection_id:string;

  reset_token:string;
}
```

A.4.16. RetireConnectionIDFrame

```
class RetireConnectionIDFrame{
  frame_type:string = "retire_connection_id";

  sequence_number:string;
}
```

A.4.17. PathChallengeFrame

```
class PathChallengeFrame{
  frame_type:string = "path_challenge";

  data?:string;
}
```

A.4.18. PathResponseFrame

```
class PathResponseFrame{
  frame_type:string = "patch_response";

  data?:string;
}
```

A.4.19. ConnectionCloseFrame

raw_error_code is the actual, numerical code. This is useful because some error types are spread out over a range of codes (e.g., QUIC's crypto_error).

```
type ErrorSpace = "transport" | "application";

class ConnectionCloseFrame{
    frame_type:string = "connection_close";

    error_space:ErrorSpace;
    error_code:TransportError | ApplicationError | number;
    raw_error_code:number;
    reason:string;

    trigger_frame_type?:string; // For known frame types, the appropriate "frame_type" string. For unknown frame types, the hex encoded identifier value
}
```

A.4.20. UnknownFrame

```
class UnknownFrame{
    frame_type:string = "unknown";
    raw_frame_type:number;

    raw?:string; // hex encoded
}
```

A.4.21. TransportError

```
enum TransportError {
    no_error,
    internal_error,
    server_busy,
    flow_control_error,
    stream_limit_error,
    stream_state_error,
    final_size_error,
    frame_encoding_error,
    transport_parameter_error,
    protocol_violation,
    invalid_migration,
    crypto_buffer_exceeded
}
```

A.4.22. CryptoError

These errors are defined in the TLS document as "A TLS alert is turned into a QUIC connection error by converting the one-byte alert description into a QUIC error code. The alert description is added to 0x100 to produce a QUIC error code from the range reserved for CRYPTO_ERROR."

This approach maps badly to a pre-defined enum. As such, we define the crypto_error string as having a dynamic component here, which should include the hex-encoded value of the TLS alert description.

```
enum CryptoError {
    crypto_error_{TLS_ALERT}
}
```

Appendix B. HTTP/3 data field definitions

B.1. HTTP/3 Frames

```
type HTTP3Frame = DataFrame | HeadersFrame | PriorityFrame |
CancelPushFrame | SettingsFrame | PushPromiseFrame | GoAwayFrame |
MaxPushIDFrame | DuplicatePushFrame | ReservedFrame | UnknownFrame;
```

B.1.1. DataFrame

```
class DataFrame{
    frame_type:string = "data";

    raw?:string; // hex encoded
}
```

B.1.2. HeadersFrame

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"}]
```

```
class HeadersFrame{
    frame_type:string = "header";
    headers:Array<HTTPHeader>;
}

class HTTPHeader {
    name:string;
    value:string;
```

```
   }
```

B.1.3. CancelPushFrame

```
class CancelPushFrame{
    frame_type:string = "cancel_push";
    push_id:string;
}
```

B.1.4. SettingsFrame

```
class SettingsFrame{
    frame_type:string = "settings";
    settings:Array<Setting>;
}

class Setting{
    name:string;
    value:string;
}
```

B.1.5. PushPromiseFrame

```
class PushPromiseFrame{
    frame_type:string = "push_promise";
    push_id:string;

    headers:Array<HTTPHeader>;
}
```

B.1.6. GoAwayFrame

```
class GoAwayFrame{
    frame_type:string = "goaway";
    stream_id:string;
}
```

B.1.7. MaxPushIDFrame

```
class MaxPushIDFrame{
    frame_type:string = "max_push_id";
    push_id:string;
}
```

B.1.8. DuplicatePushFrame

```
class DuplicatePushFrame{
    frame_type:string = "duplicate_push";
    push_id:string;
}
```

B.1.9. ReservedFrame

```
class ReservedFrame{
    frame_type:string = "reserved";
}
```

B.1.10. UnknownFrame

HTTP/3 re-uses QUIC's UnknownFrame definition, since their values and usage overlaps.

B.2. ApplicationError

```
enum ApplicationError{
    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
}
```

Appendix C. QPACK DATA type definitions

C.1. QPACK Instructions

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

```
type QPackInstruction = SetDynamicTableCapacityInstruction |
InsertWithNameReferenceInstruction |
InsertWithoutNameReferenceInstruction | DuplicateInstruction |
HeaderAcknowledgementInstruction | StreamCancellationInstruction |
InsertCountIncrementInstruction;
```

C.1.1. SetDynamicTableCapacityInstruction

```
class SetDynamicTableCapacityInstruction {
    instruction_type:string = "set_dynamic_table_capacity";

    capacity:number;
}
```

C.1.2. InsertWithNameReferenceInstruction

```
class InsertWithNameReferenceInstruction {
    instruction_type:string = "insert_with_name_reference";

    table_type:"static"|"dynamic";

    name_index:number;

    huffman_encoded_value:boolean;
    value_length:number;
    value:string;
}
```

C.1.3. InsertWithoutNameReferenceInstruction

```
class InsertWithoutNameReferenceInstruction {
    instruction_type:string = "insert_without_name_reference";

    huffman_encoded_name:boolean;
    name_length:number;
    name:string;

    huffman_encoded_value:boolean;
    value_length:number;
    value:string;
}
```

C.1.4. DuplicateInstruction

```
class DuplicateInstruction {
    instruction_type:string = "duplicate";

    index:number;
}
```

C.1.5. HeaderAcknowledgementInstruction

```
   class HeaderAcknowledgementInstruction {
       instruction_type:string = "header_acknowledgement";

       stream_id:string;
   }
```

C.1.6. StreamCancellationInstruction

```
   class StreamCancellationInstruction {
       instruction_type:string = "stream_cancellation";

       stream_id:string;
   }
```

C.1.7. InsertCountIncrementInstruction

```
   class InsertCountIncrementInstruction {
       instruction_type:string = "insert_count_increment";

       increment:number;
   }
```

C.2. QPACK Header compression

```
type QPackHeaderBlockRepresentation = IndexedHeaderField |
LiteralHeaderFieldWithName | LiteralHeaderFieldWithoutName;
```

C.2.1. IndexedHeaderField

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

class IndexedHeaderField {
    header_field_type:string = "indexed_header";

    table_type:"static"|"dynamic"; // MUST be "dynamic" if is_post_base
is true
    index:number;

    is_post_base?:boolean = false; // to represent the "indexed header
field with post-base index" header field type
}
```

C.2.2. LiteralHeaderFieldWithName

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

```
class LiteralHeaderFieldWithName {
    header_field_type:string = "literal_with_name";

    preserve_literal:boolean; // the 3rd "N" bit
    table_type:"static"|"dynamic"; // MUST be "dynamic" if is_post_base
is true
    name_index:number;

    huffman_encoded_value:boolean;
    value_length:number;
    value:string;

    is_post_base?:boolean = false; // to represent the "Literal header
field with post-base name reference" header field type
}
```

C.2.3. LiteralHeaderFieldWithoutName

```
class LiteralHeaderFieldWithoutName {
    header_field_type:string = "literal_without_name";

    preserve_literal:boolean; // the 3rd "N" bit

    huffman_encoded_name:boolean;
    name_length:number;
    name:string;

    huffman_encoded_value:boolean;
    value_length:number;
    value:string;
}
```

C.2.4. QPackHeaderBlockPrefix

```
class QPackHeaderBlockPrefix {
    required_insert_count:number;
    sign_bit:boolean;
    delta_base:number;
}
```

Appendix D. Change Log

D.1. Since draft-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)

Appendix E. Design Variations

TBD

Appendix F. Acknowledgements

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

Author's Address

Robin Marx
Hasselt University

Email: robin.marx@uhasselt.be