QUIC R. Marx
Internet-Draft Hasselt University
Intended status: Standards Track 2 November 2020
Expires: 6 May 2021

# QUIC and HTTP/3 event definitions for qlog
draft-marx-qlog-event-definitions-quic-h3-02

## 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



## 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 the QUIC and HTTP/3 protocols. This document is based on draft-29 of the QUIC and HTTP/3 I-Ds QUIC-TRANSPORT [QUIC-HTTP] and draft-16 of the QPACK I-D [QUIC-QPACK].

Feedback and discussion welcome at https://github.com/quiclog/internet-drafts (https://github.com/quiclog/internet-drafts). 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/ (https://github.com/quiclog/qlog/).

### 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].

The examples and data definitions in ths document are expressed in a custom data definition language, inspired by JSON and TypeScript, and described in [QLOG-MAIN].

## 2. Overview

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

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

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 schema document.

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

## 2.1. Importance

Many of the events defined in this document 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. This is done to reduce the amount of unique event definitions, as reflecting each possible protocol event as a separate qlog entity would cause an explosion of event types. Similarly, we prevent logging duplicate packet data as much as possible. As such, especially packet header value updates are split out into separate events (for example spin_bit_updated, connection_id_updated), as they are expected to change sparingly.

Consequently, many events that can be directly inferred from data on the wire (for example flow control limit changes) if the implementation is bug-free, are currently not explicitly defined as stand-alone events. Exceptions can be made for common events that benefit from being easily identifiable or individually logged (for example the "packets_acked" event). This can in turn give rise to separate events logging similar data, where it is not always clear which event should be logged (for example the separate "connection_started" event, whereas the more general "connection_state_updated" event also allows indicating that a connection was started).

To aid in this decision making, 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 for example frame-level details in the "Core" events due to performance or privacy 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_acked" events instead.

Finally, for event types who’s data (partially) overlap with other event types’ definitions, where necessary this document includes guidance on which to use in specific situations.

### 2.2. 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, or other member fields of 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 qlog events in a trace are typically associated with a single connection, it is unclear how to log these events.

Ideally, implementers SHOULD create a separate, individual "endpoint-level" trace file (or group_id value), not associated with a specific connection (for example a "server.qlog" or group_id = "client"), and log all events that do not belong to a single connection to this grouping trace. However, this is not always practical, depending on the implementation. Because the semantics of most of these events are well-defined in the protocols and because they are difficult to mis-interpret as belonging to a connection, implementers MAY choose to 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 retry in the same trace, while on the server they would most likely be logged in separate traces. Servers can take extra efforts (and keep additional state) to keep these events combined in a single trace however (for example by also matching connections on their four-tuple instead of just the connection ID).

## 4. QUIC and HTTP/3 fields

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

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

When the qlog "group_id" field is used, it is recommended to use QUIC's Original Destination Connection ID (ODCID, the CID chosen by the client when first contacting the server), as this is the only value that does not change over the course of the connection and can be used to link more advanced QUIC packets (e.g., Retry, Version Negotiation) to a given connection. Similarly, the ODCID should be used 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 ODCID abcd1234).

### 4.1. Raw packet and frame information

While qlog is a more high-level logging format, it also allows the inclusion of most raw wire image information, such as byte lengths and even raw byte values. This can be useful when for example investigating or tuning packetization behaviour or determining encoding/framing overheads. However, these fields are not always necessary and can take up considerable space if logged for each packet or frame. As such, they are grouped in a separate optional field called "raw" of type RawInfo (where applicable).

```
class RawInfo {
    length?:uint64; // full packet/frame length, including header and AEAD authentication tag lengths (where applicable)
    payload_length?:uint64; // length of the packet/frame payload, excluding AEAD tag. For many control frames, this will have a value of zero

    data?:bytes; // full packet/frame contents, including header and AEAD authentication tag (where applicable)
}
```

Note: QUIC packets always include an AEAD authentication tag at the end. As this tag is always the same size for a given connection (it depends on the used TLS cipher), we do not have a separate "aead_tag_length" field here. Instead, this field is reflected in "transport:parameters_set" and can be logged only once.

Note: There is intentionally no explicit header_length field in RawInfo. QUIC and HTTP/3 use many Variable-Length Integer Encoded (VLIE) values in their packet and frame headers, which are of a dynamic length. Note too that because of this, we cannot deterministically reconstruct the header encoding/length from qlog data, as implementations might not necessarily employ the most efficient VLIE scheme for all values. As such, it is typically easier to log just the total packet/frame length and the payload length. The header length can be calculated by tools as:

For QUIC packets: header_length = length - payload_length - aead_tag_length

For QUIC and HTTP/3 frames: header_length = length - payload_length

For UDP datagrams: header_length = length - payload_length

Note: In some cases, the length fields are also explicitly reflected inside of frame/packet headers. For example, the QUIC STREAM frame has a "length" field indicating its payload size. Similarly, all HTTP/3 frames include their explicit payload lengths in the frame header. Finally, the QUIC Long Header has a "length" field which is equal to the payload length plus the packet number length. 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 QUIC and HTTP/3 specifications to qlog, making it easier for users to interpret.

Note: as described in [QLOG-MAIN], the RawInfo:data field can be truncated for privacy or security purposes (for example excluding payload data). In this case, the length properties should still indicate the non-truncated lengths.

## 5. QUIC event definitions

Each subheading in this section is a qlog event category, while each sub-subheading is a qlog event type. Concretely, for the following two items, we have the category "connectivity" and event type "server_listening", resulting in a concatenated qlog "name" field value of "connectivity:server_listening".

### 5.1. connectivity

#### 5.1.1. server_listening

Importance: Extra

Emitted when the server starts accepting connections.

Data:

```
{
    ip_v4?: IPAddress,
    ip_v6?: IPAddress,
    port_v4?: uint32,
    port_v6?: uint32,

    retry_required?:boolean // the server will always answer client initials with a retry (no 1-RTT connection setups by choice)
}
```

Note: some QUIC stacks do not handle sockets directly and are thus unable to log IP and/or port information.

#### 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?: "v4" | "v6",
    src_ip?: IPAddress,
    dst_ip?: IPAddress,

    protocol?: string, // transport layer protocol (default "QUIC")
    src_port?: uint32,
    dst_port?: uint32,

    src_cid?: bytes,
    dst_cid?: bytes,

}
```

Note: some QUIC stacks do not handle sockets directly and are thus unable to log IP and/or port information.

### 5.1.3. connection_closed

Importance: Base

Used for logging when a connection was closed, typically when an error or timeout occurred. Note that this event has overlap with connectivity:connection_state_updated, as well as the CONNECTION_CLOSE frame. However, in practice, when analyzing large deployments, it can be useful to have a single event representing a connection_closed event, which also includes an additional reason field to provide additional information. Additionally, it is useful to log closures due to timeouts, which are difficult to reflect using the other options.

In QUIC there are two main connection-closing error categories: connection and application errors. They have well-defined error codes and semantics. Next to these however, there can be internal errors that occur that may or may not get mapped to the official error codes in implementation-specific ways. As such, multiple error codes can be set on the same event to reflect this.

```
{
    owner?:"local"|"remote", // which side closed the connection

    connection_code?:TransportError | CryptoError | uint32,
    application_code?:ApplicationError | uint32,
    internal_code?:uint32,

    reason?:string
}
```

Triggers: * clean * handshake_timeout * idle_timeout * error // this is called the "immediate close" in the QUIC specification * stateless_reset * version_mismatch * application // for example HTTP/3's GOAWAY frame

### 5.1.4. connection_id_updated

Importance: Base

This event is emitted when either party updates their current Connection ID. As this typically happens only sparingly over the course of a connection, this event allows loggers to be more efficient than logging the observed CID with each packet in the .header field of the "packet_sent" or "packet_received" events.

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:

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

    old?:bytes,
    new?:bytes,
}
```

### 5.1.5. 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
}
```

### 5.1.6. connection_retried

TODO

### 5.1.7. connection_state_updated

Importance: Base

This event is used to track progress through QUIC's complex handshake and connection close procedures. It is intended to provide exhaustive options to log each state individually, but also provides a more basic, simpler set for implementations less interested in tracking each smaller state transition. As such, users should not expect to see -all- these states reflected in all qlogs and implementers should focus on support for the SimpleConnectionState set.

Data: ~~~ { old?: ConnectionState | SimpleConnectionState, new: ConnectionState | SimpleConnectionState }

enum ConnectionState { attempted, // initial sent/received peer_validated, // peer address validated by: client sent Handshake packet OR client used CONNID chosen by the server. transport-draft-32, section-8.1 handshake_started, early_write, // 1 RTT can be sent, but handshake isn't done yet handshake_complete, // TLS handshake complete: Finished received and sent. tls-draft-32, section-4.1.1 handshake_confirmed, // HANDSHAKE_DONE sent/received (connection is now "active", 1RTT can be sent). tls-draft-32, section-4.1.2 closing, draining, // connection_close sent/received closed // draining period done, connection state discarded }

enum SimpleConnectionState { attempted, handshake_started, handshake_confirmed, closed } ~~~

These states correspond to the following transitions for both client and server:

*Client:*

* send initial

  - state = attempted

* get initial

  - state = validated _(not really "needed" at the client, but somewhat useful to indicate progress nonetheless)_

* get first Handshake packet

  - state = handshake_started

* get Handshake packet containing ServerFinished

  - state = handshake_complete

* send ClientFinished

  - state = early_write (1RTT can now be sent)

* get HANDSHAKE_DONE

  - state = handshake_confirmed

*Server:*

* get initial

  - state = attempted

* send initial _(don't think this needs a separate state, since some handshake will always be sent in the same flight as this?)_

* send handshake EE, CERT, CV, ...

  - state = handshake_started

* send ServerFinished

  - state = early_write (1RTT can now be sent)

* get first handshake packet / something using a server-issued CID of min length

  - state = validated

* get handshake packet containing ClientFinished

  - state = handshake_complete

* send HANDSHAKE_DONE

  - state = handshake_confirmed

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. Similarly, a state of "closing" or "draining" corresponds to the connection_closed event.

#### 5.1.8. 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?:bytes,
    new:bytes,
    generation?:uint32 // needed for 1RTT key updates
}
```

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?:bytes,
    generation?:uint32 // needed for 1RTT key updates
}
```

Triggers:

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

* "remote_update"

* "local_update"

## 5.3. transport

### 5.3.1. version_information

Importance: Core

QUIC endpoints each have their own list of of QUIC versions they support. The client uses the most likely version in their first initial. If the server does support that version, it replies with a version_negotiation packet, containing supported versions. From this, the client selects a version. This event aggregates all this information in a single event type. It also allows logging of supported versions at an endpoint without actual version negotiation needing to happen.

Data:

```
{
    server_versions?:Array<bytes>,
    client_versions?:Array<bytes>,
    chosen_version?:bytes
}
```

Intended use:

* When sending an initial, the client logs this event with client_versions and chosen_version set

* Upon receiving a client initial with a supported version, the server logs this event with server_versions and chosen_version set

* Upon receiving a client initial with an unsupported version, the server logs this event with server_versions set and client_versions to the single-element array containing the client's attempted version. The absence of chosen_version implies no overlap was found.

* Upon receiving a version negotiation packet from the server, the client logs this event with client_versions set and server_versions to the versions in the version negotiation packet and chosen_version to the version it will use for the next initial packet

### 5.3.2. alpn_information

Importance: Core

QUIC implementations each have their own list of application level protocols and versions thereof they support. The client includes a list of their supported options in its first initial as part of the TLS Application Layer Protocol Negotiation (alpn) extension. If there are common option(s), the server chooses the most optimal one and communicates this back to the client. If not, the connection is closed.

Data:

```
{
    server_alpns?:Array<string>,
    client_alpns?:Array<string>,
    chosen_alpn?:string
}
```

Intended use:

* When sending an initial, the client logs this event with client_alpns set

* When receiving an initial with a supported alpn, the server logs this event with server_alpns set, client_alpns equalling the client-provided list, and chosen_alpn to the value it will send back to the client.

* When receiving an initial with an alpn, the client logs this event with chosen_alpn to the received value.

* Alternatively, a client can choose to not log the first event, but wait for the receipt of the server initial to log this event with both client_alpns and chosen_alpn set.

#### 5.3.3. parameters_set

Importance: Core

This event groups settings from several different sources (transport parameters, 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.

In the case of connection resumption and 0-RTT, some of the server's parameters are stored up-front at the client and used for the initial connection startup. They are later updated with the server's reply. In these cases, utilize the separate "parameters_restored" event to indicate the initial values, and this event to indicate the updated values, as normal.

Data:

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

    resumption_allowed?:boolean, // valid session ticket was received
    early_data_enabled?:boolean, // early data extension was enabled on the TLS layer
    tls_cipher?:string, // (e.g., "AES_128_GCM_SHA256")
    aead_tag_length?:uint8, // depends on the TLS cipher, but it's easier to be explicit. Default value is 16

    // transport parameters from the TLS layer:
    original_destination_connection_id?:bytes,
    initial_source_connection_id?:bytes,
    retry_source_connection_id?:bytes,
    stateless_reset_token?:Token,
    disable_active_migration?:boolean,

    max_idle_timeout?:uint64,
    max_udp_payload_size?:uint32,
    ack_delay_exponent?:uint16,
    max_ack_delay?:uint16,
    active_connection_id_limit?:uint32,

    initial_max_data?:uint64,
    initial_max_stream_data_bidi_local?:uint64,
    initial_max_stream_data_bidi_remote?:uint64,
    initial_max_stream_data_uni?:uint64,
    initial_max_streams_bidi?:uint64,
    initial_max_streams_uni?:uint64,

    preferred_address?:PreferredAddress
}

interface PreferredAddress {
    ip_v4:IPAddress,
    ip_v6:IPAddress,

    port_v4:uint16,
    port_v6:uint16,

    connection_id:bytes,
    stateless_reset_token:Token
}
```

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.

### 5.3.4. parameters_restored

Importance: Base

When using QUIC 0-RTT, clients are expected to remember and restore the server's transport parameters from the previous connection. This event is used to indicate which parameters were restored and to which values when utilizing 0-RTT. Note that not all transport parameters should be restored (many are even prohibited from being re-utilized). The ones listed here are the ones expected to be useful for correct 0-RTT usage.

Data:

```
{
    disable_active_migration?:boolean,

    max_idle_timeout?:uint64,
    max_udp_payload_size?:uint32,
    active_connection_id_limit?:uint32,

    initial_max_data?:uint64,
    initial_max_stream_data_bidi_local?:uint64,
    initial_max_stream_data_bidi_remote?:uint64,
    initial_max_stream_data_uni?:uint64,
    initial_max_streams_bidi?:uint64,
    initial_max_streams_uni?:uint64,
}
```

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

### 5.3.5. packet_sent

Importance: Core

Data:

```
{
    header:PacketHeader,

    frames?:Array<QuicFrame>, // see appendix for the definitions

    is_coalesced?:boolean, // default value is false

    retry_token?:Token, // only if header.packet_type === retry

    stateless_reset_token?:bytes, // only if header.packet_type === stateless_reset. Is always 128 bits in length.

    supported_versions:Array<bytes>, // only if header.packet_type === version_negotiation

    raw?:RawInfo,
    datagram_id?:uint32
}
```

Note: We do not explicitly log the encryption_level or packet_number_space: the header.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

Note: for more details on "datagram_id", see Section 5.3.10. It is only needed when keeping track of packet coalescing.

### 5.3.6. packet_received

Importance: Core

Data:

```
{
    header:PacketHeader,

    frames?:Array<QuicFrame>, // see appendix for the definitions

    is_coalesced?:boolean,

    retry_token?:Token, // only if header.packet_type === retry

    stateless_reset_token?:bytes, // only if header.packet_type === stateless_reset. Is always 128 bits in length.

    supported_versions:Array<bytes>, // only if header.packet_type === version_negotiation

    raw?:RawInfo,
    datagram_id?:uint32
}
```

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

Triggers:

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

Note: for more details on "datagram_id", see Section 5.3.10. It is only needed when keeping track of packet coalescing.

### 5.3.7. packet_dropped

Importance: Base

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

Data:

```
{
    header?:PacketHeader, // primarily packet_type should be filled here, as other fields might not be parseable

    raw?:RawInfo,
    datagram_id?:uint32
}
```

For this event, the "trigger" field SHOULD be set (for example to one of the values below), as this helps tremendously in debugging.

Triggers:

* "key_unavailable"

* "unknown_connection_id"

* "header_parse_error"

* "payload_decrypt_error"

* "protocol_violation"

* "dos_prevention"

* "unsupported_version"

* "unexpected_packet"

* "unexpected_source_connection_id"

* "unexpected_version"

* "duplicate"

* "invalid_initial"

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 more in Section 3.

Note: for more details on "datagram_id", see Section 5.3.10. It is only needed when keeping track of packet coalescing.

### 5.3.8. 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:

```
{
    header?:PacketHeader, // primarily packet_type and possible packet_number should be filled here, as other elements might not be available yet

    raw?:RawInfo,
    datagram_id?:uint32
}
```

Note: for more details on "datagram_id", see Section 5.3.10. It is only needed when keeping track of packet coalescing.

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

### 5.3.9. packets_acked

Importance: Extra

This event is emitted when a (group of) sent packet(s) is acknowledged by the remote peer _for the first time_. This information could also be deduced from the contents of received ACK frames. However, ACK frames require additional processing logic to determine when a given packet is acknowledged for the first time, as QUIC uses ACK ranges which can include repeated ACKs. Additionally, this event can be used by implementations that do not log frame contents.

Data: ~~~ { packet_number_space?:PacketNumberSpace,

packet_numbers?:Array<uint64> } ~~~

Note: if packet_number_space is omitted, it assumes the default value of PacketNumberSpace.application_data, as this is by far the most prevalent packet number space a typical QUIC connection will use.

### 5.3.10. 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?:uint16, // to support passing multiple at once
    raw?:Array<RawInfo>, // RawInfo:length field indicates total length of the datagrams, including UDP header length

    datagram_ids?:Array<uint32>
}
```

Note: QUIC itself does not have a concept of a "datagram_id". This field is a purely qlog-specific construct to allow tracking how multiple QUIC packets are coalesced inside of a single UDP datagram, which is an important optimization during the QUIC handshake. For this, implementations assign a (per-endpoint) unique ID to each datagram and keep track of which packets were coalesced into the same datagram. As packet coalescing typically only happens during the handshake (as it requires at least one long header packet), this can be done without much overhead.

### 5.3.11. 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?:uint16, // to support passing multiple at once
    raw?:Array<RawInfo>, // RawInfo:length field indicates total length of the datagrams, including UDP header length

    datagram_ids?:Array<uint32>
}
```

Note: for more details on "datagram_ids", see Section 5.3.10.

### 5.3.12. 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:

```
{
    raw?:RawInfo
}
```

### 5.3.13. 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:uint64,
    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.

#### 5.3.14. 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.

Note: in some implementations, it can be difficult to log frames directly, even when using packet_sent and packet_received events. For these cases, this event also contains the direct packet_number field, which can be used to more explicitly link this event to the packet_sent/received events.

Data:

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

    packet_number?:uint64
}
```

#### 5.3.15. data_moved

Importance: Base

Used to indicate when data moves between the different layers (for example passing from HTTP/3 to QUIC stream buffers and vice versa) or between HTTP/3 and the actual user application on top (for example a browser engine). This helps make clear the flow of data, how long data remains in various buffers and the overheads introduced by individual layers.

For example, this helps make clear whether received data on a QUIC stream is moved to the HTTP layer immediately (for example per received packet) or in larger batches (for example, all QUIC packets are processed first and afterwards the HTTP layer reads from the streams with newly available data). This in turn can help identify bottlenecks or scheduling problems.

Data:

```
{
    stream_id?:uint64,
    offset?:uint64,
    length?:uint64, // byte length of the moved data

    from?:string, // typically: use either of "application","http","transport"
    to?:string, // typically: use either of "application","http","transport"

    data?:bytes // raw bytes that were transferred
}
```

Note: we do not for example use a "direction" field (with values "up" and "down") to specify the data flow. This is because in some optimized implementations, data might skip some individual layers. Additionally, using explicit "from" and "to" fields is more flexible and allows the definition of other conceptual "layers" (for example to indicate data from QUIC CRYPTO frames being passed to a TLS library ("security") or from HTTP/3 to QPACK ("qpack")).

Note: this event type is part of the "transport" category, but really spans all the different layers. This means we have a few leaky abstractions here (for example, the stream_id or stream offset might not be available at some logging points, or the raw data might not be in a byte-array form). In these situations, implementers can decide to define new, in-context fields to aid in manual debugging.

### 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?:uint16, // in amount of packets
    time_threshold?:float, // as RTT multiplier
    timer_granularity?:uint16, // in ms
    initial_rtt?:float, // in ms

    // congestion control, Appendix B.1.
    max_datagram_size?:uint32, // in bytes // Note: this could be updated after pmtud
    initial_congestion_window?:uint64, // in bytes
    minimum_congestion_window?:uint32, // in bytes // Note: this could change when max_datagram_size changes
    loss_reduction_factor?:float,
    persistent_congestion_threshold?:uint16 // 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?:float, // in ms or us, depending on the overarching qlog's configuration
    smoothed_rtt?:float, // in ms or us, depending on the overarching qlog's configuration
    latest_rtt?:float, // in ms or us, depending on the overarching qlog's configuration
    rtt_variance?:float, // in ms or us, depending on the overarching qlog's configuration

    pto_count?:uint16,

    // Congestion control, Appendix B.2.
    congestion_window?:uint64, // in bytes
    bytes_in_flight?:uint64,

    ssthresh?:uint64, // in bytes

    // qlog defined
    packets_in_flight?:uint64, // sum of all packet number spaces

    pacing_rate?:uint64 // 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

Data:

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

The "trigger" field 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

#### 5.4.4. loss_timer_updated

Importance: Extra

This event is emitted when a recovery loss timer changes state. The three main event types are:

* set: the timer is set with a delta timeout for when it will trigger next

* expired: when the timer effectively expires after the delta timeout

* cancelled: when a timer is cancelled (e.g., all outstanding packets are acknowledged, start idle period)

Note: to indicate an active timer's timeout update, a new "set" event is used.

Data:

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

    event_type:"set"|"expired"|"cancelled",

    delta?:float // if event_type === "set": delta time in ms or us (see configuration) from this event's timestamp until when the timer will trigger
}
```

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?

TODO: read up on the loss detection logic in draft-27 onward and see if this suffices

### 5.4.5. packet_lost

Importance: Core

This event is emitted when a packet is deemed lost by loss detection.

Data:

```
{
    header?:PacketHeader, // should include at least the packet_type and packet_number

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

For this event, the "trigger" field SHOULD be set (for example to one of the values below), as this helps tremendously in debugging.

Triggers:

* "reordering_threshold",

* "time_threshold"

* "pto_expired" // draft-23 section 5.3.1, MAY

### 5.4.6. 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?:uint64, // from SETTINGS_MAX_HEADER_LIST_SIZE
    max_table_capacity?:uint64, // from SETTINGS_QPACK_MAX_TABLE_CAPACITY
    blocked_streams_count?:uint64, // from SETTINGS_QPACK_BLOCKED_STREAMS

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

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.

### 6.1.2. parameters_restored

Importance: Base

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

Data:

```
{
    max_header_list_size?:uint64,
    max_table_capacity?:uint64,
    blocked_streams_count?:uint64
}
```

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

### 6.1.3. 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.

Data:

```
{
    stream_id:uint64,

    owner?:"local"|"remote"

    old?:StreamType,
    new:StreamType,

    associated_push_id?:uint64 // only when new == "push"
}

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

### 6.1.4. 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.

Data:

```
{
    stream_id:uint64,
    length?:uint64, // payload byte length of the frame
    frame:HTTP3Frame, // see appendix for the definitions,

    raw?:RawInfo
}
```

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.

### 6.1.5. 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.

Data:

```
{
    stream_id:uint64,
    length?:uint64, // payload byte length of the frame
    frame:HTTP3Frame, // see appendix for the definitions,

    raw?:RawInfo
}
```

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.

### 6.1.6. 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?:uint64,
    stream_id?:uint64, // 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",

    dynamic_table_capacity?:uint64,
    dynamic_table_size?:uint64, // effective current size, sum of all the entries

    known_received_count?:uint64,
    current_insert_count?:uint64
}
```

### 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:uint64,

    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 inserted or evicted from QPACK's dynamic table.

Data:

```
{
    owner:"local" | "remote", // local = the encoder's dynamic table. remote = the decoder's dynamic table

    update_type:"inserted"|"evicted",

    entries:Array<DynamicTableEntry>
}

class DynamicTableEntry {
    index:uint64;
    name?:string | bytes;
    value?:string | bytes;
}
```

### 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?:uint64,

    headers?:Array<HTTPHeader>,

    block_prefix:QPackHeaderBlockPrefix,
    header_block:Array<QPackHeaderBlockRepresentation>,

    length?:uint32,
    raw?:bytes
}
```

### 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?:uint64,

    headers?:Array<HTTPHeader>,

    block_prefix:QPackHeaderBlockPrefix,
    header_block:Array<QPackHeaderBlockRepresentation>,

    length?:uint32,
    raw?:bytes
}
```

### 6.2.6. 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.

Data:

```
{
    instruction:QPackInstruction // see appendix for the definitions,

    length?:uint32,
    raw?:bytes
}
```

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_parsed

Importance: Base

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

Data:

```
{
    instruction:QPackInstruction // see appendix for the definitions,

    length?:uint32,
    raw?:bytes
}
```

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. Generic events and Simulation indicators

### 7.1. generic

The main goal of the events in this category is to allow implementations to fully replace their existing text-based logging by qlog. This is done by providing events to log generic strings for typical well-known logging levels (error, warning, info, debug, verbose).

### 7.1.1. error

Importance: Core

Used to log details of an internal error. For errors that effectively lead to the closure of a QUIC connection, it is recommended to use transport:connection_closed instead.

Data:

```
{
    code?:uint32,
    message?:string
}
```

### 7.1.2. warning

Importance: Base

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

Data:

```
{
    code?:uint32,
    message?:string
}
```

### 7.1.3. info

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
}
```

### 7.1.4. debug

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
}
```

#### 7.1.5. verbose

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
}
```

### 7.2. simulation

When evaluating a protocol evaluation, one typically sets up a series of interoperability or benchmarking tests, in which the test situations can change over time. For example, the network bandwidth or latency can vary during the test, or the network can be fully disable for a short time. In these setups, it is useful to know when exactly these conditions are triggered, to allow for proper correlation with other events.

#### 7.2.1. scenario

Importance: Extra

Used to specify which specific scenario is being tested at this particular instance. This could also be reflected in the top-level qlog’s "summary" or "configuration" fields, but having a separate event allows easier aggregation of several simulations into one trace.

```
{
    name?:string,
    details?:any
}
```

#### 7.2.2. marker

Importance: Extra

Used to indicate when 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).

```
{
    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", Work in Progress, Internet-Draft, draft-marx-qlog-main-schema-02, 2 November 2020, <https://tools.ietf.org/html/draft-marx-qlog-main-schema-02>.

[QUIC-HTTP]
Bishop, M., Ed., "Hypertext Transfer Protocol Version 3 (HTTP/3)", Work in Progress, Internet-Draft, draft-ietf-quic-http-32, 1 October 2020, <https://tools.ietf.org/html/draft-ietf-quic-http-32>.

[QUIC-QPACK]
Frindell, A., Ed., "QPACK: Header Compression for HTTP/3", Work in Progress, Internet-Draft, draft-ietf-quic-qpack-19, 20 October 2020, <https://tools.ietf.org/html/draft-ietf-quic-qpack-19>.

[QUIC-TRANSPORT]
Iyengar, J., Ed. and M. Thomson, Ed., "QUIC: A UDP-Based Multiplexed and Secure Transport", Work in Progress, Internet-Draft, draft-ietf-quic-transport-32, 1 October 2020, <https://tools.ietf.org/html/draft-ietf-quic-transport-32>.

## 10.2. Informative References

[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>.

# Appendix A. QUIC data field definitions

## A.1. IPAddress

```
class IPAddress : string | bytes;

// an IPAddress can either be a "human readable" form (e.g., "127.0.0.1" for v4 or "2001:0db8:85a3:0000:0000:8a2e:0370:7334" for v6) or use a raw byte-form (as the string forms can be ambiguous)
```

## A.2. PacketType

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

## A.3. PacketNumberSpace

```
enum PacketNumberSpace {
    initial,
    handshake,
    application_data
}
```

## A.4. PacketHeader

```
class PacketHeader {
    // Note: short vs long header is implicit through PacketType

    packet_type: PacketType;
    packet_number: uint64;

    flags?: uint8; // the bit flags of the packet headers (spin bit, key update bit, etc. up to and including the packet number length bits if present) interpreted as a single 8-bit integer

    token?:Token; // only if packet_type == initial

    length?: uint16, // only if packet_type == initial || handshake || 0RTT. Signifies length of the packet_number plus the payload.

    // only if present in the header
    // if correctly using transport:connection_id_updated events,
    // dcid can be skipped for 1RTT packets
    version?: bytes; // e.g., "ff00001d" for draft-29
    scil?: uint8;
    dcil?: uint8;
    scid?: bytes;
    dcid?: bytes;
}
```

### A.5. Token

```
class Token {
    type?:"retry"|"resumption"|"stateless_reset";

    length?:uint32; // byte length of the token
    data?:bytes; // raw byte value of the token

    details?:any; // decoded fields included in the token (typically: peer's IP address, creation time)
}
```

The token carried in an Initial packet can either be a retry token from a Retry packet, a stateless reset token from a Stateless Reset packet or one originally provided by the server in a NEW_TOKEN frame used when resuming a connection (e.g., for address validation purposes). Retry and resumption tokens typically contain encoded metadata to check the token's validity when it is used, but this metadata and its format is implementation specific. For that, this field includes a general-purpose "details" field.

### A.6. 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.7. QUIC Frames

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

### A.7.1. PaddingFrame

In QUIC, PADDING frames are simply identified as a single byte of value 0. As such, each padding byte could be theoretically interpreted and logged as an individual PaddingFrame.

However, as this leads to heavy logging overhead, implementations SHOULD instead emit just a single PaddingFrame and set the payload_length property to the amount of PADDING bytes/frames included in the packet.

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

    length?:uint32; // total frame length, including frame header
    payload_length?:uint32;
}
```

### A.7.2. PingFrame

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

    length?:uint32; // total frame length, including frame header
    payload_length?:uint32;
}
```

### A.7.3. AckFrame

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

    ack_delay?:float; // in ms

    // 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<[uint64, uint64]|[uint64]>;

    // ECN (explicit congestion notification) related fields (not always present)
    ect1?:uint64;
    ect0?:uint64;
    ce?:uint64;

    length?:uint32; // total frame length, including frame header
    payload_length?:uint32;
}
```

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.7.4. ResetStreamFrame

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

    stream_id:uint64;
    error_code:ApplicationError | uint32;
    final_size:uint64; // in bytes

    length?:uint32; // total frame length, including frame header
    payload_length?:uint32;
}
```

### A.7.5. StopSendingFrame

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

    stream_id:uint64;
    error_code:ApplicationError | uint32;

    length?:uint32; // total frame length, including frame header
    payload_length?:uint32;
}
```

### A.7.6. CryptoFrame

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

    offset:uint64;
    length:uint64;

    payload_length?:uint32;
}
```

### A.7.7. NewTokenFrame

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

  token:Token
}
```

### A.7.8. StreamFrame

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

    stream_id:uint64;

    // These two MUST always be set
    // If not present in the Frame type, log their default values
    offset:uint64;
    length:uint64;

    // 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?:bytes;
}
```

### A.7.9. MaxDataFrame

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

  maximum:uint64;
}
```

### A.7.10. MaxStreamDataFrame

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

  stream_id:uint64;
  maximum:uint64;
}
```

### A.7.11. MaxStreamsFrame

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

  stream_type:string = "bidirectional" | "unidirectional";
  maximum:uint64;
}
```

### A.7.12. DataBlockedFrame

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

  limit:uint64;
}
```

### A.7.13. StreamDataBlockedFrame

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

  stream_id:uint64;
  limit:uint64;
}
```

### A.7.14. StreamsBlockedFrame

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

  stream_type:string = "bidirectional" | "unidirectional";
  limit:uint64;
}
```

### A.7.15. NewConnectionIDFrame

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

  sequence_number:uint32;
  retire_prior_to:uint32;

  connection_id_length?:uint8;
  connection_id:bytes;

  stateless_reset_token?:Token;
}
```

### A.7.16. RetireConnectionIDFrame

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

  sequence_number:uint32;
}
```

### A.7.17. PathChallengeFrame

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

  data?:bytes; // always 64-bit
}
```

### A.7.18. PathResponseFrame

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

  data?:bytes; // always 64-bit
}
```

### A.7.19.  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 | uint32;
    raw_error_code?:uint32;
    reason?:string;

    trigger_frame_type?:uint64 | string; // For known frame types, the appropriate "frame_type" string. For unknown frame types, the hex encoded identifier value
}
```

### A.7.20.  HandshakeDoneFrame

```
class HandshakeDoneFrame{
  frame_type:string = "handshake_done";
}
```

### A.7.21.  UnknownFrame

```
class UnknownFrame{
    frame_type:string = "unknown";
    raw_frame_type:uint64;

    raw_length?:uint32;
    raw?:bytes;
}
```

### A.7.22.  TransportError

```
enum TransportError {
    no_error,
    internal_error,
    connection_refused,
    flow_control_error,
    stream_limit_error,
    stream_state_error,
    final_size_error,
    frame_encoding_error,
    transport_parameter_error,
    connection_id_limit_error,
    protocol_violation,
    invalid_token,
    application_error,
    crypto_buffer_exceeded
}
```

### A.7.23. 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?:bytes;
}
```

### 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:uint64;
}
```

### 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:uint64;

    headers:Array<HTTPHeader>;
}
```

### B.1.6. GoAwayFrame

```
class GoAwayFrame{
    frame_type:string = "goaway";
    stream_id:uint64;
}
```

### B.1.7. MaxPushIDFrame

```
class MaxPushIDFrame{
    frame_type:string = "max_push_id";
    push_id:uint64;
}
```

### B.1.8. DuplicatePushFrame

```
class DuplicatePushFrame{
    frame_type:string = "duplicate_push";
    push_id:uint64;
}
```

### 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:uint32;
}
```

### C.1.2. InsertWithNameReferenceInstruction

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

    table_type:"static"|"dynamic";

    name_index:uint32;

    huffman_encoded_value:boolean;

    value_length?:uint32;
    value?:string;
}
```

### C.1.3. InsertWithoutNameReferenceInstruction

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

    huffman_encoded_name:boolean;

    name_length?:uint32;
    name?:string;

    huffman_encoded_value:boolean;

    value_length?:uint32;
    value?:string;
}
```

### C.1.4. DuplicateInstruction

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

    index:uint32;
}
```

### C.1.5. HeaderAcknowledgementInstruction

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

    stream_id:uint64;
}
```

### C.1.6. StreamCancellationInstruction

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

    stream_id:uint64;
}
```

### C.1.7. InsertCountIncrementInstruction

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

    increment:uint32;
}
```

## 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:uint32;

    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:uint32;

    huffman_encoded_value:boolean;
    value_length?:uint32;
    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?:uint32;
    name?:string;

    huffman_encoded_value:boolean;
    value_length?:uint32;
    value?:string;
}
```

### C.2.4.  QPackHeaderBlockPrefix

```
class QPackHeaderBlockPrefix {
    required_insert_count:uint32;
    sign_bit:boolean;
    delta_base:uint32;
}
```

# Appendix D.  Change Log

## D.1.  Since draft-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)

### D.2. 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 Marten Seemann, Jana Iyengar, Brian Trammell, Dmitri Tikhonov, Stephen Petrides, Jari Arkko, Marcus Ihlar, Victor Vasiliev, Mirja Kuehlewind, Jeremy Laine, Kazu Yamamoto, Christian Huitema, and Lucas Pardue for their feedback and suggestions.

## Author's Address

Robin Marx
Hasselt University

Email: robin.marx@uhasselt.be