Workgroup: QUIC
Internet-Draft:
draft-ietf-quic-qlog-quic-events-04
Published: 13 February 2023
Intended Status: Standards Track
Expires: 17 August 2023
Authors: R. Marx, Ed. L. Niccolini, Ed. M. Seemann, Ed.
Akamai Meta Protocol Labs
L. Pardue, Ed.
Cloudflare

QUIC event definitions for qlog

Abstract

This document describes concrete qlog event definitions and their metadata for QUIC 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 QUIC; see [QUIC-TRANSPORT], [QUIC-RECOVERY], and [QUIC-TLS].

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

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

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

1.1. Notational Conventions

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

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

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

2. Overview

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

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

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

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).

2.1. Raw packet and frame information

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

Note: As QUIC uses trailers in packets, packet header_lengths can be calculated as:

header_length = length - payload_length - aead_tag_length

For UDP datagrams, the calculation is simpler:

header_length = length - payload_length

Note: In some cases, the length fields are also explicitly reflected inside of packet headers. For example, the QUIC STREAM frame has a "length" field indicating its payload size. Similarly, 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 specifications to qlog, making it easier for users to interpret.

2.2. 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).

3. QUIC Event Overview

QUIC connections consist of different phases and interaction events. In order to model this, QUIC event types are divided into general categories: connectivity (Section 4), security (Section 6), transport Section 5, and recovery Section 7.

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

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

Name value	Importance	Definition
connectivity:server_listening	Extra	Section 4.1
connectivity:connection_started	Base	Section 4.2
connectivity:connection_closed	Base	Section 4.3
connectivity:connection_id_updated	Base	Section 4.4
connectivity:spin_bit_updated	Base	Section 4.5
connectivity:connection_state_updated	Base	Section 4.6
connectivity:mtu_updated	Extra	Section 4.8
transport:version_information	Core	Section 5.1
transport:alpn_information	Core	Section 5.2
transport:parameters_set	Core	Section 5.3
transport:parameters_restored	Base	Section 5.4
transport:packet_sent	Core	Section 5.5
transport:packet_received	Core	Section 5.6
transport:packet_dropped	Base	Section 5.7
transport:packet_buffered	Base	Section 5.8

Name value	Importance	Definition
transport:packets_acked	Extra	Section 5.9
transport:datagrams_sent	Extra	Section 5.10
transport:datagrams_received	Extra	Section 5.11
transport:datagram_dropped	Extra	Section 5.12
transport:stream_state_updated	Base	Section 5.13
transport:frames_processed	Extra	Section 5.14
transport:data_moved	Base	Section 5.15
security:key_updated	Base	Section 6.1
security:key_discarded	Base	Section 6.2
recovery:parameters_set	Base	Section 7.1
recovery:metrics_updated	Core	Section 7.2
recovery:congestion_state_updated	Base	Section 7.3
recovery:loss_timer_updated	Extra	Section 7.4
recovery:packet_lost	Core	Section 7.5
recovery:marked_for_retransmit	Extra	Section 7.6

Table 1: QUIC Events

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

```
QuicEvents = ConnectivityServerListening /
             ConnectivityConnectionStarted /
             ConnectivityConnectionClosed /
             ConnectivityConnectionIDUpdated /
             ConnectivitySpinBitUpdated /
             ConnectivityConnectionStateUpdated /
             ConnectivityMTUUpdated /
             SecurityKeyUpdated / SecurityKeyDiscarded /
             TransportVersionInformation / TransportALPNInformation /
             TransportParametersSet / TransportParametersRestored /
             TransportPacketSent / TransportPacketReceived /
             TransportPacketDropped / TransportPacketBuffered /
             TransportPacketsAcked / TransportDatagramsSent /
             TransportDatagramsReceived / TransportDatagramDropped /
             TransportStreamStateUpdated / TransportFramesProcessed /
             TransportDataMoved /
             RecoveryParametersSet / RecoveryMetricsUpdated /
             RecoveryCongestionStateUpdated /
             RecoveryLossTimerUpdated /
             RecoveryPacketLost

$ProtocolEventBody /= QuicEvents
```

Figure 1: QuicEvents definition and ProtocolEventBody extension

4. Connectivity events

4.1. server_listening

Importance: Extra

Emitted when the server starts accepting connections.

Definition:

```
ConnectivityServerListening = {
    ? ip_v4: IPAddress
    ? ip_v6: IPAddress
    ? port_v4: uint16
    ? port_v6: uint16

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

Figure 2: ConnectivityServerListening definition

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

4.2. 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.

Definition:

```
ConnectivityConnectionStarted = {
    ? ip_version: IPVersion
    src_ip: IPAddress
    dst_ip: IPAddress

    ; transport layer protocol
    ? protocol: text .default "QUIC"
    ? src_port: uint16
    ? dst_port: uint16

    ? src_cid: ConnectionID
    ? dst_cid: ConnectionID
}
```

Figure 3: ConnectivityConnectionStarted definition

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

4.3. 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.

Definition:

```
ConnectivityConnectionClosed = {
    ; which side closed the connection
    ? owner: Owner

    ? connection_code: TransportError / CryptoError / uint32
    ? application_code: $ApplicationError / uint32
    ? internal_code: uint32

    ? reason: text
    ? trigger:
        "clean" /
        "handshake_timeout" /
        "idle_timeout" /
        ; this is called the "immediate close" in the QUIC RFC
        "error" /
        "stateless_reset" /
        "version_mismatch" /
        ; for example HTTP/3's GOAWAY frame
        "application"
}
```

Figure 4: ConnectivityConnectionClosed definition

4.4. 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 endpoint applying the new id. As such, when the endpoint receives a new connection id from the peer, it will see the dst_ fields are set. When the endpoint updates its own connection id (e.g., NEW_CONNECTION_ID frame), it logs the src_ fields.

Definition:

```
ConnectivityConnectionIDUpdated = {
    owner: Owner

    ? old: ConnectionID
    ? new: ConnectionID
}
```

Figure 5: ConnectivityConnectionIDUpdated definition

4.5. 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.

Definition:

```
ConnectivitySpinBitUpdated = {
    state: bool
}
```

Figure 6: ConnectivitySpinBitUpdated definition

4.6. 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.

Definition:

```
ConnectivityConnectionStateUpdated = {
    ? old: ConnectionState / SimpleConnectionState
    new: ConnectionState / SimpleConnectionState
}

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

SimpleConnectionState =
    "attempted" /
    "handshake_started" /
    "handshake_confirmed" /
    "closed"
```

Figure 7: ConnectivityConnectionStateUpdated definition

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 *(TODO 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.

4.7. 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

4.8. mtu_updated

Importance: Extra

```
ConnectivityMTUUpdated = {
  ? old: uint16
  new: uint16

  ; at some point, MTU discovery stops, as a "good enough"
  ; packet size has been found
  ? done: bool .default false
}
```

Figure 8: ConnectivityMTUUpdated definition

This event indicates that the estimated Path MTU was updated. This happens as part of the Path MTU discovery process.

5. Transport events

5.1. 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.

Definition:

```
TransportVersionInformation = {
    ? server_versions: [+ QuicVersion]
    ? client_versions: [+ QuicVersion]
    ? chosen_version: QuicVersion
}
```

Figure 9: TransportVersionInformation definition

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

5.2. 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.

Definition:

```
TransportALPNInformation = {
    ? server_alpns: [* text]
    ? client_alpns: [* text]
    ? chosen_alpn: text
}
```

Figure 10: TransportALPNInformation definition

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. 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.

Definition:

```
TransportParametersSet = {
    ? owner: Owner

    ; true if valid session ticket was received
    ? resumption_allowed: bool

    ; true if early data extension was enabled on the TLS layer
    ? early_data_enabled: bool

    ; e.g., "AES_128_GCM_SHA256"
    ? tls_cipher: text

    ; depends on the TLS cipher, but it's easier to be explicit.
    ; in bytes
    ? aead_tag_length: uint8 .default 16

    ; transport parameters from the TLS layer:
    ? original_destination_connection_id: ConnectionID
    ? initial_source_connection_id: ConnectionID
    ? retry_source_connection_id: ConnectionID
    ? stateless_reset_token: StatelessResetToken
    ? disable_active_migration: bool

    ? 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
}

PreferredAddress = {
    ip_v4: IPAddress
    ip_v6: IPAddress

    port_v4: uint16
    port_v6: uint16

    connection_id: ConnectionID
    stateless_reset_token: StatelessResetToken
}
```

Figure 11: TransportParametersSet definition

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.

5.4. 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.

Definition:

```
TransportParametersRestored = {
    ? disable_active_migration: bool

    ? 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
}
```

Figure 12: TransportParametersRestored definition

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

5.5. packet_sent

Importance: Core

Definition:

```
TransportPacketSent = {
    header: PacketHeader

    ? frames: [* $QuicFrame]

    ? is_coalesced: bool .default false

    ; only if header.packet_type === "retry"
    ? retry_token: Token

    ; only if header.packet_type === "stateless_reset"
    ; is always 128 bits in length.
    ? stateless_reset_token: StatelessResetToken

    ; only if header.packet_type === "version_negotiation"
    ? supported_versions: [+ QuicVersion]

    ? raw: RawInfo
    ? datagram_id: uint32

    ? is_mtu_probe_packet: bool .default false

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

Figure 13: TransportPacketSent definition

Note: The encryption_level and packet_number_space are not logged explicitly: the header.packet_type specifies this by inference (assuming correct implementation)

Note: for more details on "datagram_id", see Section 5.10. It is only needed when keeping track of packet coalescing.

5.6. packet_received

Importance: Core

Definition:

```
TransportPacketReceived = {
    header: PacketHeader

    ? frames: [* $QuicFrame]

    ? is_coalesced: bool .default false

    ; only if header.packet_type === "retry"
    ? retry_token: Token

    ; only if header.packet_type === "stateless_reset"
    ; Is always 128 bits in length.
    ? stateless_reset_token: StatelessResetToken

    ; only if header.packet_type === "version_negotiation"
    ? supported_versions: [+ QuicVersion]

    ? raw: RawInfo
    ? datagram_id: uint32

    ? trigger:
        ; if packet was buffered because
        ; it couldn't be decrypted before
        "keys_available"
}
```

Figure 14: TransportPacketReceived definition

Note: The encryption_level and packet_number_space are not logged explicitly: the header.packet_type specifies this by inference (assuming correct implementation)

Note: for more details on "datagram_id", see Section 5.10. It is only needed when keeping track of packet coalescing.

5.7. packet_dropped

Importance: Base

This event indicates a QUIC-level packet was dropped.

The trigger field indicates a general reason category for dropping the packet, while the details field can contain additional implementation-specific information.

Definition:

```
TransportPacketDropped = {
    ; Primarily packet_type should be filled here,
    ; as other fields might not be decrypteable or parseable
    ? header: PacketHeader

    ? raw: RawInfo
    ? datagram_id: uint32

    ? details: {* text => any}
    ? trigger:
        "internal_error" /
        "rejected" /
        "unsupported" /
        "invalid" /
        "connection_unknown" /
        "decryption_failure" /
        "general"
}
```

Figure 15: TransportPacketDropped definition

Some example situations for each of the trigger categories include:

* internal_error: not initialized, out of memory

* rejected: limits reached, DDoS protection, unwilling to track more paths, duplicate packet

* unsupported: unknown or unsupported version. See also Section 2.2.

* invalid: packet parsing or validation error

* connection_unknown: packet does not relate to a known connection or Connection ID

* decryption_failure: decryption key was unavailable, decryption failed

* general: situations not clearly covered in the other categories

For more details on "datagram_id", see Section 5.10.

5.8. 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 only the full packet contents can be logged when it was parsed in a packet_received event.

Definition:

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

    ? raw: RawInfo
    ? datagram_id: uint32

    ? trigger:
        ; indicates the parser cannot keep up, temporarily buffers
        ; packet for later processing
        "backpressure" /
        ; if packet cannot be decrypted because the proper keys were
        ; not yet available
        "keys_unavailable"
}
```

Figure 16: TransportPacketBuffered definition

Note: for more details on "datagram_id", see Section 5.10. It is only needed when keeping track of packet coalescing.

5.9. 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.

Definition:

```
TransportPacketsAcked = {
    ? packet_number_space: PacketNumberSpace

    ? packet_numbers: [+ uint64]
}
```

Figure 17: TransportPacketsAcked definition

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.

5.10. datagrams_sent

Importance: Extra

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

Definition:

```
TransportDatagramsSent = {
    ; to support passing multiple at once
    ? count: uint16

    ; The RawInfo fields do not include the UDP headers,
    ; only the UDP payload
    ? raw: [+ RawInfo]

    ? datagram_ids: [+ uint32]
}
```

Figure 18: TransportDatagramsSent definition

Since QUIC implementations rarely control UDP logic directly, the raw data excludes UDP-level headers in all fields.

The "datagram_id" is a qlog-specific concept to allow tracking of QUIC packet coalescing inside UDP datagrams. Implementations can assign a per-endpoint unique ID to each datagram, and reflect this in other events to track QUIC packets through processing steps.

5.11. datagrams_received

Importance: Extra

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

Definition:

```
TransportDatagramsReceived = {
    ; to support passing multiple at once
    ? count: uint16

    ; The RawInfo fields do not include the UDP headers,
    ; only the UDP payload
    ? raw: [+ RawInfo]

    ? datagram_ids: [+ uint32]
}
```

Figure 19: TransportDatagramsReceived definition

For more details on "datagram_ids", see Section 5.10.

5.12. datagram_dropped

Importance: Extra

When a UDP-level datagram is dropped. This is typically done if it does not contain a valid QUIC packet. If it does, but the QUIC packet is dropped for other reasons, packet_dropped (Section 5.7) should be used instead.

Definition:

```
TransportDatagramDropped = {
    ; The RawInfo fields do not include the UDP headers,
    ; only the UDP payload
    ? raw: RawInfo
}
```

Figure 20: TransportDatagramDropped definition

5.13. 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.

Definition:

```
StreamType = "unidirectional" / "bidirectional"

TransportStreamStateUpdated = {
    stream_id: uint64

    ; mainly useful when opening the stream
    ? stream_type: StreamType

    ? old: StreamState
    new: StreamState

    ? stream_side: "sending" / "receiving"
}

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:
    ; memory actually freed
    "destroyed"
```

Figure 21: TransportStreamStateUpdated definition

Note: QUIC implementations **SHOULD** mainly log the simplified bidirectional (HTTP/2-alike) stream states (e.g., idle, open, closed) instead of the more fine-grained 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.

5.14. 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). Implementations have 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, these events are aggregated into 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. 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.

Definition:

```
TransportFramesProcessed = {
    frames: [* $QuicFrame]

    ? packet_number: uint64
}
```

Figure 22: TransportFramesProcessed definition

5.15. data_moved

Importance: Base

Used to indicate when data moves between the different layers (for example passing from the application protocol (e.g., HTTP) to QUIC stream buffers and vice versa) or between the application protocol (e.g., HTTP) 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 application protocol immediately (for example per received packet) or in larger batches (for example, all QUIC packets are processed first and afterwards the application layer reads from the streams with newly available data). This in turn can help identify bottlenecks or scheduling problems.

Definition:

```
TransportDataMoved = {
    ? stream_id: uint64
    ? offset: uint64

    ; byte length of the moved data
    ? length: uint64

    ? from: "user" / "application" / "transport" / "network" / text
    ? to: "user" / "application" / "transport" / "network" / text

    ? raw: RawInfo
}
```

Figure 23: TransportDataMoved definition

6. Security Events

6.1. 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

Definition:

```
SecurityKeyUpdated = {
    key_type: KeyType

    ? old: hexstring
    new: hexstring

    ; needed for 1RTT key updates
    ? generation: uint32

    ? trigger:
        ; (e.g., initial, handshake and 0-RTT keys
        ; are generated by TLS)
        "tls" /
        "remote_update" /
        "local_update"
}
```

Figure 24: SecurityKeyUpdated definition

6.2. key_discarded

Importance: Base

Definition:

```
SecurityKeyDiscarded = {
    key_type: KeyType
    ? key: hexstring

    ; needed for 1RTT key updates
    ? generation: uint32

    ? trigger:
        ; (e.g., initial, handshake and 0-RTT keys
        ; are generated by TLS)
        "tls" /
        "remote_update" /
        "local_update"
}
```

Figure 25: SecurityKeyDiscarded definition

7. Recovery events

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).

7.1. 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.

Definition:

```
RecoveryParametersSet = {
    ; Loss detection, see recovery draft-23, Appendix A.2
    ; in amount of packets
    ? reordering_threshold: uint16

    ; as RTT multiplier
    ? time_threshold: float32

    ; in ms
    timer_granularity: uint16

    ; in ms
    ? initial_rtt:float32

    ; congestion control, Appendix B.1.
    ; in bytes. Note: this could be updated after pmtud
    ? max_datagram_size: uint32

    ; in bytes
    ? initial_congestion_window: uint64

    ; Note: this could change when max_datagram_size changes
    ; in bytes
    ? minimum_congestion_window: uint64
    ? loss_reduction_factor: float32

    ; as PTO multiplier
    ? persistent_congestion_threshold: uint16
}
```

Figure 26: RecoveryParametersSet definition

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

7.2. 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.

Definition:

```
RecoveryMetricsUpdated = {
    ; Loss detection, see recovery draft-23, Appendix A.3
    ; all following rtt fields are expressed in ms
    ? min_rtt: float32
    ? smoothed_rtt: float32
    ? latest_rtt: float32
    ? rtt_variance: float32

    ? pto_count: uint16

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

    ; in bytes
    ? ssthresh: uint64

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

    ; in bits per second
    ? pacing_rate: uint64
}
```

Figure 27: RecoveryMetricsUpdated definition

Note: to make logging easier, implementations **MAY** log values even if they are the same as previously reported values (e.g., two subsequent RecoveryMetricsUpdated 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.

7.3. 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

Definition:

```
RecoveryCongestionStateUpdated = {
    ? old: text
    new: text

    ? trigger:
        "persistent_congestion" /
        "ECN"
}
```

Figure 28: RecoveryCongestionStateUpdated definition

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 occurring (e.g., slow start is exited only when ssthresh is exceeded).

7.4. 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.

Definition:

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

    event_type: "set" / "expired" / "cancelled"

    ; if event_type === "set": delta time is in ms from
    ; this event's timestamp until when the timer will trigger
    ? delta: float32
}
```

Figure 29: RecoveryLossTimerUpdated definition

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

7.5. packet_lost

Importance: Core

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

Definition:

```
RecoveryPacketLost = {
    ; should include at least the packet_type and packet_number
    ? header: PacketHeader

    ; not all implementations will keep track of full
    ; packets, so these are optional
    ? frames: [* $QuicFrame]

    ? is_mtu_probe_packet: bool .default false

    ? trigger:
        "reordering_threshold" /
        "time_threshold" /
        ; draft-23 section 5.3.1, MAY
        "pto_expired"
}
```

Figure 30: RecoveryPacketLost definition

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

7.6. marked_for_retransmit

Importance: Extra

This event indicates which data was marked for retransmit upon detecting 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, this signal is grouped into in a single event based on existing QUIC frame definitions for all types of retransmittable data.

Implementations retransmitting full packets or frames directly can just log the constituent 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).

Definition:

```
RecoveryMarkedForRetransmit = {
    frames: [+ $QuicFrame]
}
```

Figure 31: RecoveryMarkedForRetransmit definition

8. QUIC data field definitions

8.1. QuicVersion

```
QuicVersion = hexstring
```

Figure 32: QuicVersion definition

8.2. ConnectionID

```
ConnectionID = hexstring
```

Figure 33: ConnectionID definition

8.3. Owner

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

Figure 34: Owner definition

8.4. IPAddress and IPVersion

```
; 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)
IPAddress = text / hexstring
```

Figure 35: IPAddress definition

```
IPVersion = "v4" / "v6"
```

Figure 36: IPVersion definition

8.5. PacketType

```
PacketType = "initial" / "handshake" / "0RTT" / "1RTT" / "retry" /
    "version_negotiation" / "stateless_reset" / "unknown"
```

Figure 37: PacketType definition

8.6. PacketNumberSpace

```
PacketNumberSpace = "initial" / "handshake" / "application_data"
```

Figure 38: PacketNumberSpace definition

8.7. PacketHeader

```
PacketHeader = {
    packet_type: PacketType
    ; only if packet_type === "initial" || "handshake" || "0RTT" ||
    ;                         "1RTT"
    ? packet_number: uint64

    ; the bit flags of the packet headers (spin bit, key update bit,
    ; etc. up to and including the packet number length bits
    ; if present
    ? flags: uint8

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

    ; only if packet_type === "initial" || "handshake" || "0RTT"
    ; Signifies length of the packet_number plus the payload
    ? length: uint16

    ; only if present in the header
    ; if correctly using transport:connection_id_updated events,
    ; dcid can be skipped for 1RTT packets
    ? version: QuicVersion
    ? scil: uint8
    ? dcil: uint8
    ? scid: ConnectionID
    ? dcid: ConnectionID
}
```

Figure 39: PacketHeader definition

8.8. Token

```
Token = {
    ? type: "retry" / "resumption"

    ; decoded fields included in the token
    ; (typically: peer's IP address, creation time)
    ? details: {
      * text => any
    }

    ? raw: RawInfo
}
```

Figure 40: Token definition

The token carried in an Initial packet can either be a retry token from a Retry 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 event includes a general-purpose "details" field.

8.9. Stateless Reset Token

```
StatelessResetToken = hexstring .size 16
```

Figure 41: Stateless Reset Token definition

The stateless reset token is carried in stateless reset packets, in transport parameters and in NEW_CONNECTION_ID frames.

8.10. KeyType

```
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"
```

Figure 42: KeyType definition

8.11. QUIC Frames

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

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

Figure 43: QuicFrame plug definition

The QUIC frame types defined in this document are as follows:

```
QuicBaseFrames /=
  PaddingFrame / PingFrame / AckFrame / ResetStreamFrame /
  StopSendingFrame / CryptoFrame / NewTokenFrame / StreamFrame /
  MaxDataFrame / MaxStreamDataFrame / MaxStreamsFrame /
  DataBlockedFrame / StreamDataBlockedFrame / StreamsBlockedFrame /
  NewConnectionIDFrame / RetireConnectionIDFrame /
  PathChallengeFrame / PathResponseFrame / ConnectionCloseFrame /
  HandshakeDoneFrame / UnknownFrame

$QuicFrame /= QuicBaseFrames
```

Figure 44: QuicBaseFrames definition

8.11.1. 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.

```
PaddingFrame = {
    frame_type: "padding"

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

Figure 45: PaddingFrame definition

8.11.2. PingFrame

```
PingFrame = {
    frame_type: "ping"

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

Figure 46: PingFrame definition

8.11.3. AckFrame

```
; either a single number (e.g., [1]) or two numbers (e.g., [1,2]).
; For two numbers:
; the first number is "from": lowest packet number in interval
; the second number is "to": up to and including the highest
; packet number in the interval
AckRange = [1*2 uint64]

AckFrame = {
    frame_type: "ack"

    ; in ms
    ? ack_delay: float32

    ; e.g., looks like [[1,2],[4,5], [7], [10,22]] serialized
    ? acked_ranges: [+ AckRange]

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

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

Figure 47: AckFrame definition

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.

8.11.4. ResetStreamFrame

```
ResetStreamFrame = {
    frame_type: "reset_stream"

    stream_id: uint64
    error_code: $ApplicationError / uint32

    ; in bytes
    final_size: uint64

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

Figure 48: ResetStreamFrame definition

8.11.5. StopSendingFrame

```
StopSendingFrame = {
    frame_type: "stop_sending"

    stream_id: uint64
    error_code: $ApplicationError / uint32

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

Figure 49: StopSendingFrame definition

8.11.6. CryptoFrame

```
CryptoFrame = {
    frame_type: "crypto"

    offset: uint64
    length: uint64

    ? payload_length: uint32
}
```

Figure 50: CryptoFrame definition

8.11.7. NewTokenFrame

```
NewTokenFrame = {
  frame_type: "new_token"

  token: Token
}
```

Figure 51: NewTokenFrame definition

8.11.8. StreamFrame

```
StreamFrame = {
    frame_type: "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: bool .default false

    ? raw: RawInfo
}
```

Figure 52: StreamFrame definition

8.11.9. MaxDataFrame

```
MaxDataFrame = {
  frame_type: "max_data"

  maximum: uint64
}
```

Figure 53: MaxDataFrame definition

8.11.10. MaxStreamDataFrame

```
MaxStreamDataFrame = {
  frame_type: "max_stream_data"

  stream_id: uint64
  maximum: uint64
}
```

Figure 54: MaxStreamDataFrame definition

8.11.11. MaxStreamsFrame

```
MaxStreamsFrame = {
  frame_type: "max_streams"

  stream_type: StreamType
  maximum: uint64
}
```

Figure 55: MaxStreamsFrame definition

8.11.12. DataBlockedFrame

```
DataBlockedFrame = {
  frame_type: "data_blocked"

  limit: uint64
}
```

Figure 56: DataBlockedFrame definition

8.11.13. StreamDataBlockedFrame

```
StreamDataBlockedFrame = {
  frame_type: "stream_data_blocked"

  stream_id: uint64
  limit: uint64
}
```

Figure 57: StreamDataBlockedFrame definition

8.11.14. StreamsBlockedFrame

```
StreamsBlockedFrame = {
  frame_type: "streams_blocked"

  stream_type: StreamType
  limit: uint64
}
```

Figure 58: StreamsBlockedFrame definition

8.11.15. NewConnectionIDFrame

```
NewConnectionIDFrame = {
  frame_type: "new_connection_id"

  sequence_number: uint32
  retire_prior_to: uint32

  ; mainly used if e.g., for privacy reasons the full
  ; connection_id cannot be logged
  ? connection_id_length: uint8
  connection_id: ConnectionID

  ? stateless_reset_token: StatelessResetToken
}
```

Figure 59: NewConnectionIDFrame definition

8.11.16. RetireConnectionIDFrame

```
RetireConnectionIDFrame = {
  frame_type: "retire_connection_id"

  sequence_number: uint32
}
```

Figure 60: RetireConnectionIDFrame definition

8.11.17. PathChallengeFrame

```
PathChallengeFrame = {
  frame_type: "path_challenge"

  ; always 64-bit
  ? data: hexstring
}
```

Figure 61: PathChallengeFrame definition

8.11.18. PathResponseFrame

```
PathResponseFrame = {
  frame_type: "path_response"

  ; always 64-bit
  ? data: hexstring
}
```

Figure 62: PathResponseFrame definition

8.11.19. ConnectionCloseFrame

The error_code_value field is the numerical value without VLIE encoding. This is useful because some error types are spread out over a range of codes (e.g., QUIC's crypto_error).

```
ErrorSpace = "transport" / "application"

ConnectionCloseFrame = {
    frame_type: "connection_close"

    ? error_space: ErrorSpace
    ? error_code: TransportError / $ApplicationError / uint32
    ? error_code_value: uint64
    ? reason: text

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

Figure 63: ConnectionCloseFrame definition

8.11.20. HandshakeDoneFrame

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

Figure 64: HandshakeDoneFrame definition

8.11.21. UnknownFrame

The frame_type_value field is the numerical value without VLIE encoding.

```
UnknownFrame = {
    frame_type: "unknown"
    frame_type_value: uint64

    ? raw: RawInfo
}
```

Figure 65: UnknownFrame definition

8.11.22. TransportError

```
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" / "key_update_error" /
    "aead_limit_reached" / "no_viable_path"
    ; there is no value to reflect CRYPTO_ERROR
    ; use the CryptoError type instead
```

Figure 66: TransportError definition

8.11.23. ApplicationError

By definition, an application error is defined by the application-level protocol running on top of QUIC (e.g., HTTP/3).

As such, it cannot be defined here directly. Applications **MAY** use the provided extension point through the use of the CDDL "socket" mechanism.

Application-level qlog definitions that wish to define new ApplicationError strings **MUST** do so by extending the $ApplicationError socket as such:

```
$ApplicationError /= "new_error_name" / "another_new_error_name"
```

8.11.24. 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, the crypto_error string is defined as having a dynamic component here, which should include the hex-encoded and zero-padded value of the TLS alert description.

```
; all strings from "crypto_error_0x100" to "crypto_error_0x1ff"
CryptoError = text .regexp "crypto_error_0x1[0-9a-f][0-9a-f]"
```

Figure 67: CryptoError definition

9. Security and Privacy Considerations

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

10. IANA Considerations

TBD

11. Normative References

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

[QLOG-MAIN] Marx, R., Niccolini, L., Seemann, M., and L. Pardue, "Main logging schema for qlog", Work in Progress, Internet-Draft, draft-ietf-quic-qlog-main-schema-04, 24 October 2022, <https://datatracker.ietf.org/doc/html/draft-ietf-quic-qlog-main-schema-04>.

[QUIC-RECOVERY] Iyengar, J., Ed. and I. Swett, Ed., "QUIC Loss Detection and Congestion Control", RFC 9002, DOI 10.17487/RFC9002, May 2021, <https://www.rfc-editor.org/rfc/rfc9002>.

[QUIC-TLS] Snijders, J., Heitz, J., Scudder, J., and A. Azimov, "Extended BGP Administrative Shutdown Communication", RFC 9003, DOI 10.17487/RFC9003, January 2021, <https://www.rfc-editor.org/rfc/rfc9003>.

[QUIC-TRANSPORT] Iyengar, J., Ed. and M. Thomson, Ed., "QUIC: A UDP-Based Multiplexed and Secure Transport", RFC 9000, DOI 10.17487/RFC9000, May 2021, <https://www.rfc-editor.org/rfc/rfc9000>.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/

RFC2119, March 1997, <https://www.rfc-editor.org/rfc/rfc2119>.

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

Appendix A. Change Log

A.1. Since draft-ietf-qlog-quic-events-03:

* Ensured consistent use of RawInfo to indicate raw wire bytes (#243)

* Renamed UnknownFrame:raw_frame_type to :frame_type_value (#54)

* Renamed ConnectionCloseFrame:raw_error_code to :error_code_value (#54)

* Changed triggers for packet_dropped (#278)

* Added entries to TransportError enum (#285)

* Changed minimum_congestion_window to uint64 (#288)

A.2. Since draft-ietf-qlog-quic-events-02:

* Renamed key_retired to key_discarded (#185)

* Added fields and events for DPLPMTUD (#135)

* Made packet_number optional in PacketHeader (#244)

* Removed connection_retried event placeholder (#255)

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

* Moved data definitions out of the appendix into separate sections

* Added overview Table of Contents

A.3. Since draft-ietf-qlog-quic-events-01:

* Added Stateless Reset Token type (#122)

A.4. Since draft-ietf-qlog-quic-events-00:

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

A.5. Since draft-marx-qlog-event-definitions-quic-h3-02:

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

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

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

* Changed to/from value options of the data_moved event

A.6. Since draft-marx-qlog-event-definitions-quic-h3-01:

Major changes:

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

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

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

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

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

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

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

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

Smaller changes:

* Merged loss_timer events into one loss_timer_updated event

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

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

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

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

*Removed push_allowed from http:parameters_set (#110)

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

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

*Added length property to PaddingFrame (#34)

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

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

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

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

*Added "packets_acked" event (#107)

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

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

A.7. Since draft-marx-qlog-event-definitions-quic-h3-00:

*Event and category names are now all lowercase

*Added many new events and their definitions

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

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

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

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

Acknowledgements

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

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

Authors' Addresses

Robin Marx (editor)
Akamai

Email: rmarx@akamai.com

Luca Niccolini (editor)
Meta

Email: lniccolini@meta.com

Marten Seemann (editor)
Protocol Labs

Email: marten@protocol.ai

Lucas Pardue (editor)
Cloudflare

Email: lucaspardue.24.7@gmail.com