QUIC
Internet-Draft
Intended status: Standards Track
Expires: 8 January 2026

R. Marx, Ed.
Akamai
L. Niccolini, Ed.
Meta
M. Seemann, Ed.

L. Pardue, Ed.
Cloudflare
7 July 2025

QUIC event definitions for qlog
draft-ietf-quic-qlog-quic-events-11

Abstract

This document describes a qlog event schema containing concrete qlog event definitions and their metadata for the core QUIC protocol and selected extensions.

Note to Readers

> Note to RFC editor: Please remove this section before publication.

Feedback and discussion are welcome at https://github.com/quicwg/qlog (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.

Copyright Notice

Table of Contents

1. Introduction

This document defines a qlog event schema (Section 8 of [QLOG-MAIN]) containing concrete events for the core QUIC protocol (see [QUIC-TRANSPORT], [QUIC-RECOVERY], and [QUIC-TLS]) and some of its extensions (see [QUIC-DATAGRAM] and [GREASEBIT]).

The event namespace with identifier quic is defined; see Section 2. In this namespace multiple events derive from the qlog abstract Event class (Section 7 of [QLOG-MAIN]), each extending the "data" field and defining their "name" field values and semantics. Some event data fields use complex data types. These are represented as enums or re-usable definitions, which are grouped together on the bottom of this document for clarity.

1.1. Use of group IDs

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).

1.2. Raw packet and frame information

QUIC packets always include an AEAD authentication tag at the end. In general, the length of the AEAD tag depends on the TLS cipher suite, although all cipher suites used in QUIC v1 use a 16 byte tag. For the purposes of calculating the lengths in fields of type RawInfo (as defined in [QLOG-MAIN]) related to QUIC packets, the AEAD tag is regarded as a trailer.

1.3. Events not belonging to a single connection

A single qlog event trace is typically associated with a single QUIC connection. However, for several types of events (for example, a Section 5.7 event with trigger value of connection_unknown), it can be impossible to tie them to a specific QUIC connection, especially on the server.

There are various ways to handle these events, each making certain tradeoffs between file size overhead, flexibility, ease of use, or ease of implementation. Some options include:

* Log them in a separate endpoint-wide trace (or use a special group_id value) not associated with a single connection.

* Log them in the most recently used trace.

* Use additional heuristics for connection identification (for example use the four-tuple in addition to the Connection ID).

* Buffer events until they can be assigned to a connection (for example for version negotiation and retry events).

1.4. 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, namespace, type, data, group_id, RawInfo, and time-related fields.

Events are defined with an importance level as described in Section 8.3 of [QLOG-MAIN].

As is the case for [QLOG-MAIN], the qlog schema definitions in this document are intentionally agnostic to serialization formats. The choice of format is an implementation decision.

2. Event Schema Definition

This document describes how the core QUIC protocol and selected extensions can be expressed in qlog using a newly defined event schema. Per the requirements in Section 8 of [QLOG-MAIN], this document registers the quic namespace. The event schema URI is urn:ietf:params:qlog:events:quic.

2.1. Draft Event Schema Identification

This section is to be removed before publishing as an RFC.

Only implementations of the final, published RFC can use the events belonging to the event schema with the URI urn:ietf:params:qlog:events:quic. Until such an RFC exists, implementations MUST NOT identify themselves using this URI.

Implementations of draft versions of the event schema MUST append the string "-" and the corresponding draft number to the URI. For example, draft 07 of this document is identified using the URI urn:ietf:params:qlog:events:quic-07.

The namespace identifier itself is not affected by this requirement.

3. QUIC Event Overview

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

Name value	Importance	Definition
quic:server_listening	Extra	Section 4.1
quic:connection_started	Base	Section 4.2
quic:connection_closed	Base	Section 4.3
quic:connection_id_updated	Base	Section 4.4
quic:spin_bit_updated	Base	Section 4.5
quic:connection_state_updated	Base	Section 4.6
quic:path_assigned	Base	Section 4.7
quic:mtu_updated	Extra	Section 4.8
quic:version_information	Core	Section 5.1
quic:alpn_information	Core	Section 5.2
quic:parameters_set	Core	Section 5.3
quic:parameters_restored	Base	Section 5.4
quic:packet_sent	Core	Section 5.5
quic:packet_received	Core	Section 5.6
quic:packet_dropped	Base	Section 5.7
quic:packet_buffered	Base	Section 5.8
quic:packets_acked	Extra	Section 5.9
quic:udp_datagrams_sent	Extra	Section 5.10
quic:udp_datagrams_received	Extra	Section 5.11
quic:udp_datagram_dropped	Extra	Section 5.12

quic:stream_state_updated	Base	Section 5.13
quic:frames_processed	Extra	Section 5.14
quic:stream_data_moved	Base	Section 5.15
quic:datagram_data_moved	Base	Section 5.16
quic:connection_data_blocked_updated	Extra	Section 5.17
quic:stream_data_blocked_updated	Extra	Section 5.18
quic:datagram_data_blocked_updated	Extra	Section 5.19
quic:migration_state_updated	Extra	Section 5.20
quic:key_updated	Base	Section 6.1
quic:key_discarded	Base	Section 6.2
quic:recovery_parameters_set	Base	Section 7.1
quic:recovery_metrics_updated	Core	Section 7.2
quic:congestion_state_updated	Base	Section 7.3
quic:loss_timer_updated	Extra	Section 7.4
quic:packet_lost	Core	Section 7.5
quic:marked_for_retransmit	Extra	Section 7.6
quic:ecn_state_updated	Extra	Section 7.7

Table 1: QUIC Events

QUIC events extend the $ProtocolEventData extension point defined in [QLOG-MAIN]. Additionally, they allow for direct extensibility by their use of per-event extension points via the $$ CDDL "group socket" syntax, as also described in [QLOG-MAIN].

```
QuicEventData = QUICServerListening /
                QUICConnectionStarted /
                QUICConnectionClosed /
                QUICConnectionIDUpdated /
                QUICSpinBitUpdated /
                QUICConnectionStateUpdated /
                QUICPathAssigned /
                QUICMTUUpdated /
                QUICVersionInformation /
                QUICALPNInformation /
                QUICParametersSet /
                QUICParametersRestored /
                QUICPacketSent /
                QUICPacketReceived /
                QUICPacketDropped /
                QUICPacketBuffered /
                QUICPacketsAcked /
                QUICUDPDatagramsSent /
                QUICUDPDatagramsReceived /
                QUICUDPDatagramDropped /
                QUICStreamStateUpdated /
                QUICFramesProcessed /
                QUICStreamDataMoved /
                QUICDatagramDataMoved /
                QUICConnectionDataBlockedUpdated /
                QUICStreamDataBlockedUpdated /
                QUICDatagramDataBlockedUpdated /
                QUICMigrationStateUpdated /
                QUICKeyUpdated /
                QUICKeyDiscarded /
                QUICRecoveryParametersSet /
                QUICRecoveryMetricsUpdated /
                QUICCongestionStateUpdated /
                QUICLossTimerUpdated /
                QUICPacketLost /
                QUICMarkedForRetransmit /
                QUICECNStateUpdated

$ProtocolEventData /= QuicEventData
```

Figure 1: QuicEventData definition and ProtocolEventData extension

The concrete QUIC event types are further defined below, their type identifier is the heading name. The subdivisions in sections on Connectivity, Security, Transport and Recovery are purely for readability.

4. Connectivity events

4.1. server_listening

Emitted when the server starts accepting connections. It has Extra importance level.

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

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

    * $$quic-serverlistening-extension
}
```

Figure 2: QUICServerListening definition

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

4.2. connection_started

The connection_started event is used for both attempting (client-perspective) and accepting (server-perspective) new connections. Note that while there is overlap with the connection_state_updated event, this event is separate event in order to capture additional data that can be useful to log. It has Base importance level.

```
QUICConnectionStarted = {
    local: PathEndpointInfo
    remote: PathEndpointInfo

    * $$quic-connectionstarted-extension
}
```

Figure 3: QUICConnectionStarted definition

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

4.3. connection_closed

The connection_closed event is used for logging when a connection was closed, typically when an error or timeout occurred. It has Base importance level.

Note that this event has overlap with the connection_state_updated event, 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 more information. Furthermore, it is useful to log closures due to timeouts or explicit application actions (such as racing multiple connections and aborting the slowest), which are difficult to reflect using the other options.

The connection_closed event is intended to be logged either when the local endpoint silently discards the connection due to an idle timeout, when a CONNECTION_CLOSE frame is sent (the connection enters the ’closing’ state on the sender side), when a CONNECTION_CLOSE frame is received (the connection enters the ’draining’ state on the receiver side) or when a Stateless Reset packet is received (the connection is discarded at the receiver side). Connectivity-related updates after this point (e.g., exiting a ’closing’ or ’draining’ state), should be logged using the connection_state_updated event instead.

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, and more fine-grained internal error codes can be reflected in the internal_code field.

If the error code does not map to a known error string, the connection_code or application_code value of "unknown" type can be used and the raw value captured in the code_bytes field; a numerical value without variable-length integer encoding.

```
QUICConnectionClosed = {

    ; which side closed the connection
    ? owner: Owner
    ? connection_code: $TransportError /
                       CryptoError
    ? application_code: $ApplicationError

    ; if connection_code or application_code === "unknown"
    ? code_bytes: uint32

    ? internal_code: uint32
    ? reason: text
    ? trigger:
        "idle_timeout" /
        "application" /
        "error" /
        "version_mismatch" /
        ; when received from peer
        "stateless_reset" /
        "aborted" /
        ; when it is unclear what triggered the CONNECTION_CLOSE
        "unspecified"

    * $$quic-connectionclosed-extension
}
```

Figure 4: QUICConnectionClosed definition

Loggers SHOULD use the most descriptive trigger for a connection_closed event that they are able to deduce. This is often clear at the peer closing the connection (and sending the CONNECTION_CLOSE), but can sometimes be more opaque at the receiving end.

4.4. connection_id_updated

The connection_id_updated event is emitted when either party updates their current Connection ID. As this typically happens only sparingly over the course of a connection, using this event is more efficient than logging the observed CID with each and every packet_sent or packet_received events. It has Base importance level.

The connection_id_updated event 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, the owner field will be "remote". When the endpoint updates its own connection ID, the owner field will be "local".

```
QUICConnectionIDUpdated = {
    owner: Owner
    ? old: ConnectionID
    ? new: ConnectionID

    * $$quic-connectionidupdated-extension
}
```

Figure 5: QUICConnectionIDUpdated definition

4.5. spin_bit_updated

The spin_bit_updated event conveys information about the QUIC latency spin bit; see Section 17.4 of [QUIC-TRANSPORT]. The event is emitted when the spin bit changes value, it SHOULD NOT be emitted if the spin bit is set without changing its value. It has Base importance level.

```
QUICSpinBitUpdated = {
    state: bool

    * $$quic-spinbitupdated-extension
}
```

Figure 6: QUICSpinBitUpdated definition

4.6. connection_state_updated

The connection_state_updated event is used to track progress through QUIC’s complex handshake and connection close procedures. It has Base importance level.

[QUIC-TRANSPORT] does not contain an exhaustive flow diagram with possible connection states nor their transitions (though some are explicitly mentioned, like the ’closing’ and ’draining’ states). As such, this document *non-exhaustively* defines those states that are most likely to be useful for debugging QUIC connections.

QUIC implementations SHOULD mainly log the simplified BaseConnectionStates, adding the more fine-grained GranularConnectionStates when more in-depth debugging is required. Tools SHOULD be able to deal with both types equally.

```
QUICConnectionStateUpdated = {
    ? old: $ConnectionState
    new: $ConnectionState

    * $$quic-connectionstateupdated-extension
}
```

```
BaseConnectionStates =
    ; Initial packet sent/received
    "attempted" /

    ; Handshake packet sent/received
    "handshake_started" /

    ; Both sent a TLS Finished message
    ; and verified the peer's TLS Finished message
    ; 1-RTT packets can be sent
    ; RFC 9001 Section 4.1.1
    "handshake_complete" /

    ; CONNECTION_CLOSE sent/received,
    ; stateless reset received or idle timeout
    "closed"

GranularConnectionStates =
    ; RFC 9000 Section 8.1
    ; client sent Handshake packet OR
    ; client used connection ID chosen by the server OR
    ; client used valid address validation token
    "peer_validated" /

    ; 1-RTT data can be sent by the server,
    ; but handshake is not done yet
    ; (server has sent TLS Finished; sometimes called 0.5 RTT data)
    "early_write" /

    ; HANDSHAKE_DONE sent/received.
    ; RFC 9001 Section 4.1.2
    "handshake_confirmed" /

    ; CONNECTION_CLOSE sent
    "closing" /

    ; CONNECTION_CLOSE received
    "draining" /

    ; draining or closing period done, connection state discarded
    "closed"

$ConnectionState /= BaseConnectionStates / GranularConnectionStates
```

Figure 7: QUICConnectionStateUpdated definition

The connection_state_changed event has some overlap with the connection_closed and connection_started events, and the handling of various frames (for example in a packet_received event). Still, it can be useful to log these logical state transitions separately, especially if they map to an internal implementation state machine, to explicitly track progress. As such, implementations are allowed to use other ConnectionState values that adhere more closely to their internal logic. Tools SHOULD be able to deal with these custom states in a similar way to the pre-defined states in this document.

4.7. path_assigned

Importance: Base

This event is used to associate a single PathID’s value with other parameters that describe a unique network path.

As described in [QLOG-MAIN], each qlog event can be linked to a single network path by means of the top-level "path" field, whose value is a PathID. However, since it can be cumbersome to encode additional path metadata (such as IP addresses or Connection IDs) directly into the PathID, this event allows such an association to happen separately. As such, PathIDs can be short and unique, and can even be updated to be associated with new metadata as the connection’s state evolves.

Definition:

```
QUICPathAssigned = {
    path_id: PathID

    ; the information for traffic going towards the remote receiver
    ? path_remote: PathEndpointInfo

    ; the information for traffic coming in at the local endpoint
    ? path_local: PathEndpointInfo

    * $$quic-pathassigned-extension
}
```

Figure 8: QUICPathAssigned definition

Choosing the different path_id values is left up to the implementation. Some options include using a uniquely incrementing integer, using the (first) Destination Connection ID associated with a path (or its sequence number), or using (a hash of) the two endpoint IP addresses.

It is important to note that the empty string ("") is a valid PathID and that it is the default assigned to events that do not explicitly set a "path" field. Put differently, the initial path of a QUIC connection on which the handshake occurs (see also Section 4.2) is implicitly associated with the PathID with value "". Associating metadata with this default path is possible by logging the QUICPathAssigned event with a value of "" for the path_id field.

As paths and their metadata can evolve over time, multiple QUICPathAssigned events can be emitted for each unique PathID. The latest event contains the most up-to-date information for that PathID. As such, the first time a PathID is seen in a QUICPathAssigned event, it is an indication that the path is created. Subsequent occurrences indicate the path is updated, while a final occurrence with both path_local and path_remote fields omitted implicitly indicates the path has been abandoned.

4.8. mtu_updated

The mtu_updated event indicates that the estimated Path MTU was updated. This happens as part of the Path MTU discovery process. It has Extra importance level.

```
QUICMTUUpdated = {
    ? old: uint32
    new: uint32

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

    * $$quic-mtuupdated-extension
}
```

Figure 9: QUICMTUUpdated definition

5. Transport events

5.1. version_information

The version_information event supports QUIC version negotiation; see Section 6 of [QUIC-TRANSPORT]. It has Core importance level.

QUIC endpoints each have their own list of QUIC versions they support. The client uses the most likely version in their first initial. If the server does not support that version, it replies with a Version Negotiation packet, which contains its supported versions. From this, the client selects a version. The version_information 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.

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

    * $$quic-versioninformation-extension
}
```

Figure 10: QUICVersionInformation 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. If the client receives a set of server_versions with no viable overlap with its own supported versions, this event should be logged without the chosen_version set

5.2. alpn_information

The alpn_information event supports Application-Layer Protocol Negotiation (ALPN) over the QUIC transport; see [RFC7301] and Section 7.4 of [QUIC-TRANSPORT]. It has Core importance level.

QUIC endpoints are configured with a list of supported ALPN identifiers. Clients send the list in a TLS ClientHello, and servers match against their list. On success, a single ALPN identifier is chosen and sent back in a TLS ServerHello. If no match is found, the connection is closed.

ALPN identifiers are byte sequences, that may be possible to present as UTF-8. The ALPNIdentifier‘ type supports either format. Implementations SHOULD log at least one format, but MAY log both or none.

```
QUICALPNInformation = {
    ? server_alpns: [* ALPNIdentifier]
    ? client_alpns: [* ALPNIdentifier]
    ? chosen_alpn: ALPNIdentifier

    * $$quic-alpninformation-extension
}

ALPNIdentifier = {
  ? byte_value: hexstring
  ? string_value: text
}
```

Figure 11: QUICALPNInformation 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

The parameters_set 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. The event has Core importance level.

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

Implementations are not required to recognize, process or support every setting/parameter received in all situations. For example, QUIC implementations MUST discard transport parameters that they do not understand Section 7.4.2 of [QUIC-TRANSPORT]. The unknown_parameters field can be used to log the raw values of any unknown parameters (e.g., GREASE, private extensions, peer-side experimentation).

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.

```
QUICParametersSet = {
    ? 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

    ; RFC9000
    ? 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
        ? unknown_parameters: [* UnknownParameter]

        ; RFC9221
        ? max_datagram_frame_size: uint64

        ; RFC9287
        ; true if present, absent or false if extension not negotiated
        ? grease_quic_bit: bool

        * $$quic-parametersset-extension
    }

    PreferredAddress = {
        ? ip_v4: IPAddress
        ? port_v4: uint16
        ? ip_v6: IPAddress
        ? port_v6: uint16
        connection_id: ConnectionID
        stateless_reset_token: StatelessResetToken
    }

    UnknownParameter = {
        id: uint64
        ? value: hexstring
    }
```

Figure 12: QUICParametersSet definition

5.4. parameters_restored

When using QUIC 0-RTT, clients are expected to remember and restore the server’s transport parameters from the previous connection. The parameters_restored event is used to indicate which parameters were restored and to which values when utilizing 0-RTT. It has Base importance level.

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.

```
QUICParametersRestored = {

    ; RFC9000
    ? 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

    ; RFC9221
    ? max_datagram_frame_size: uint64

    ; RFC9287
    ; can only be restored at the client.
    ; servers MUST NOT restore this parameter!
    ? grease_quic_bit: bool

    * $$quic-parametersrestored-extension
}
```

Figure 13: QUICParametersRestored definition

5.5. packet_sent

The packet_sent event indicates a QUIC-level packet was sent. It has Core importance level.

```
QUICPacketSent = {
    header: PacketHeader
    ? frames: [* $QuicFrame]

    ; 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:
      ; RFC 9002 Section 6.1.1
      "retransmit_reordered" /
      ; RFC 9002 Section 6.1.2
      "retransmit_timeout" /
      ; RFC 9002 Section 6.2.4
      "pto_probe" /
      ; RFC 9002 6.2.3
      "retransmit_crypto" /
      ; needed for some CCs to figure out bandwidth allocations
      ; when there are no normal sends
      "cc_bandwidth_probe"

    * $$quic-packetsent-extension
}
```

Figure 14: QUICPacketSent definition

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

The datagram_id field is used to track packet coalescing, see Section 5.10.

5.6. packet_received

The packet_received event indicates a QUIC-level packet was received. It has Core importance level.

```
QUICPacketReceived = {
    header: PacketHeader
    ? frames: [* $QuicFrame]

    ; 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"

    * $$quic-packetreceived-extension
}
```

Figure 15: QUICPacketReceived definition

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

The datagram_id field is used to track packet coalescing, see Section 5.10.

5.7. packet_dropped

The packet_dropped event indicates a QUIC-level packet was dropped. It has Base importance level.

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

```
QUICPacketDropped = {

    ; 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" /
        "duplicate" /
        "connection_unknown" /
        "decryption_failure" /
        "key_unavailable" /
        "general"

    * $$quic-packetdropped-extension
}
```

Figure 16: QUICPacketDropped 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 1.3.

* invalid: packet parsing or validation error

* duplicate: duplicate packet

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

* decryption_failure: decryption failed

* key_unavailable: decryption key was unavailable

* general: situations not clearly covered in the other categories

The datagram_id field is used to track packet coalescing, see Section 5.10.

5.8. packet_buffered

The packet_buffered 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. The event has Base importance level.

```
QUICPacketBuffered = {

    ; 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"

    * $$quic-packetbuffered-extension
}
```

Figure 17: QUICPacketBuffered definition

The datagram_id field is used to track packet coalescing, see Section 5.10.

5.9. packets_acked

The packets_acked event is emitted when a (group of) sent packet(s) is acknowledged by the remote peer _for the first time_. It has Extra importance level.

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.

```
QUICPacketsAcked = {
    ? packet_number_space: $PacketNumberSpace
    ? packet_numbers: [+ uint64]

    * $$quic-packetsacked-extension
}
```

Figure 18: QUICPacketsAcked definition

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

5.10. udp_datagrams_sent

The datagrams_sent event indicates when one or more UDP-level datagrams are passed to the underlying network socket. This is useful for determining how QUIC packet buffers are drained to the OS. The event has Extra importance level.

```
QUICUDPDatagramsSent = {

    ; to support passing multiple at once
    ? count: uint16

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

    ; ECN bits in the IP header
    ; if not set, defaults to the value used on the last
    ; QUICDatagramsSent event
    ? ecn: [+ ECN]

    ? datagram_ids: [+ uint32]

    * $$quic-udpdatagramssent-extension
}
```

Figure 19: QUICUDPDatagramsSent definition

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

Multiple QUIC packets can be coalesced in a single UDP datagram, especially during the handshake (see Section 12.2 of [QUIC-TRANSPORT]). However, neither QUIC nor UDP themselves provide an explicit mechanism to track this behaviour. To make it possible for implementations to track coalescing across packet-level and datagram-level qlog events, this document defines a qlog-specific mechanism for tracking coalescing across packet-level and datagram-level qlog events: a "datagram identifier" carried in datagram_id fields. qlog implementations that want to track coalescing can use this mechanism, where multiple events sharing the same datagram_id indicate they were coalesced in the same UDP datagram. The selection of specific and locally-unique datagram_id values is an implementation choice.

5.11. udp_datagrams_received

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. The event has Extra importance level.

```
QUICUDPDatagramsReceived = {

    ; to support passing multiple at once
    ? count: uint16

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

    ; ECN bits in the IP header
    ; if not set, defaults to the value on the last
    ; QUICDatagramsReceived event
    ? ecn: [+ ECN]

    ? datagram_ids: [+ uint32]

    * $$quic-udpdatagramsreceived-extension
}
```

Figure 20: QUICUDPDatagramsReceived definition

The datagram_ids field is used to track packet coalescing, see Section 5.10.

5.12. udp_datagram_dropped

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, the packet_dropped event (Section 5.7) should be used instead. The event has Extra importance level.

```
QUICUDPDatagramDropped = {

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

    * $$quic-udpdatagramdropped-extension
}
```

Figure 21: QUICUDPDatagramDropped definition

5.13. stream_state_updated

The stream_state_updated event is emitted whenever the internal state of a QUIC stream is updated; see Section 3 of [QUIC-TRANSPORT]. 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. The event has Base importance level.

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

QUICStreamStateUpdated = {
    stream_id: uint64

    ; mainly useful when opening the stream
    ? stream_type: StreamType
    ? old: $StreamState
    new: $StreamState
    ? stream_side: "sending" /
                   "receiving"

    * $$quic-streamstateupdated-extension
}

BaseStreamStates =  "idle" /
                    "open" /
                    "closed"

GranularStreamStates =
    ; bidirectional stream states, RFC 9000 Section 3.4.
    "half_closed_local" /
    "half_closed_remote" /
    ; sending-side stream states, RFC 9000 Section 3.1.
    "ready" /
    "send" /
    "data_sent" /
    "reset_sent" /
    "reset_received" /
    ; receive-side stream states, RFC 9000 Section 3.2.
    "receive" /
    "size_known" /
    "data_read" /
    "reset_read" /
    ; both-side states
    "data_received" /
    ; qlog-defined: memory actually freed
    "destroyed"

$StreamState /= BaseStreamStates / GranularStreamStates
```

Figure 22: QUICStreamStateUpdated definition

QUIC implementations SHOULD mainly log the simplified (HTTP/2-alike) BaseStreamStates instead of the more fine-grained GranularStreamStates. These latter ones are mainly for more in-depth debugging. Tools SHOULD be able to deal with both types equally.

5.14. frames_processed

The frame_processed event is intended to prevent a large proliferation of specific purpose events (e.g., packets_acknowledged, flow_control_updated, stream_data_received). It has Extra importance level.

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.

The frame_processed 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).

The packet_received event can convey all constituent frames. It is not expected that the frames_processed event will also be used for a redundant purpose. Rather, implementations can use this event to avoid having to log full packets or to convey extra information about when frames are processed (for example, if frame processing is deferred for any reason).

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

In some implementations, it can be difficult to log frames directly, even when using packet_sent and packet_received events. For these cases, the frames_processed event also contains the packet_numbers field, which can be used to more explicitly link this event to the packet_sent/received events. The field is an array, which supports using a single frames_processed event for multiple frames received over multiple packets. To map between frames and packets, the position and order of entries in the frames and packet_numbers is used. If the optional packet_numbers field is used, each frame MUST have a corresponding packet number at the same index.

```
QUICFramesProcessed = {
    frames: [* $QuicFrame]
    ? packet_numbers: [* uint64]

    * $$quic-framesprocessed-extension
}
```

Figure 23: QUICFramesProcessed definition

For example, an instance of the frames_processed event that represents four STREAM frames received over two packets would have the fields serialized as:


```
"frames":[
  {"frame_type":"stream","stream_id":0,"offset":0,"length":500},
  {"frame_type":"stream","stream_id":0,"offset":500,"length":200},
  {"frame_type":"stream","stream_id":1,"offset":0,"length":300},
  {"frame_type":"stream","stream_id":1,"offset":300,"length":50}
  ],
"packet_numbers":[
  1,
  1,
  2,
  2
]
```


5.15. stream_data_moved

The stream_data_moved event is used to indicate when QUIC stream data moves between the different layers. This helps make clear the flow of data, how long data remains in various buffers, and the overheads introduced by individual layers. The event has Base importance level.

This event relates to stream data only. There are no packet or frame headers and length values in the length or raw fields MUST reflect that.

For example, it can be useful to understand when data moves from an application protocol (e.g., HTTP) to QUIC stream buffers and vice versa.

The stream_data_moved event can provide insight into 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 can help identify bottlenecks, flow control issues, or scheduling problems.

The additional_info field supports optional logging of information related to the stream state. For example, an application layer that moves data into transport and simultaneously ends the stream, can log fin_set. As another example, a transport layer that has received an instruction to reset a stream can indicate this to the application layer using reset_stream. In both cases, the length-carrying fields (length or raw) can be omitted or contain zero values.

This event is only for data in QUIC streams. For data in QUIC Datagram Frames, see the datagram_data_moved event defined in Section 5.16.

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

    ; byte length of the moved data
    ? length: uint64

    ? from: $DataLocation
    ? to: $DataLocation

    ? additional_info: $DataMovedAdditionalInfo

    ? raw: RawInfo

    * $$quic-streamdatamoved-extension
}

$DataLocation /=  "application" /
                  "transport" /
                  "network"

$DataMovedAdditionalInfo /= "fin_set" /
                            "stream_reset"
```

Figure 24: QUICStreamDataMoved definition

5.16. datagram_data_moved

The datagram_data_moved event is used to indicate when QUIC Datagram Frame data (see [RFC9221]) moves between the different layers. This helps make clear the flow of data, how long data remains in various buffers, and the overheads introduced by individual layers. The event has Base importance level.

This event relates to datagram data only. There are no packet or frame headers and length values in the length or raw fields MUST reflect that.

For example, passing from the application protocol (e.g., WebTransport) to QUIC Datagram Frame buffers and vice versa.

The datagram_data_moved event can provide insight into whether received data in a QUIC Datagram Frame 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 all Datagrams at once). This can help identify bottlenecks, flow control issues, or scheduling problems.

This event is only for data in QUIC Datagram Frames. For data in QUIC streams, see the stream_data_moved event defined in Section 5.15.

```
QUICDatagramDataMoved = {
    ; byte length of the moved data
    ? length: uint64
    ? from: $DataLocation
    ? to: $DataLocation
    ? raw: RawInfo

    * $$quic-datagramdatamoved-extension
}
```

Figure 25: QUICDatagramDataMoved definition

5.17. connection_data_blocked_updated

The connection_blocked_updated event is used to indicate when the QUIC connection becomes blocked or unblocked for sending data. When a connection is "blocked", data can't be sent in streams and/or datagrams until the blocking reason has been resolved. The event has Extra importance level.

Use the stream_blocked_updated or datagram_blocked_updated event to provide more fine-grained information for individual data types.

```
QUICConnectionDataBlockedUpdated = {
    ? old: $BlockedState
    new: $BlockedState

    ? reason: $BlockedReason
}

$BlockedState /= "blocked" /
               "unblocked"

$BlockedReason /= "scheduling" /
                  "pacing" /
                  "amplification_protection" /
                  "congestion_control" /
                  "connection_flow_control" /
                  "stream_flow_control" /
                  "stream_id" /
                  "application"
```

Figure 26: QUICConnectionDataBlockedUpdated definition

5.18. stream_data_blocked_updated

The stream_data_blocked_updated event is used to indicate when a QUIC stream becomes blocked or unblocked for sending. The event has Extra importance level.

```
QUICStreamDataBlockedUpdated = {
    ? old: $BlockedState
    new: $BlockedState

    stream_id: uint64

    ? reason: $BlockedReason
}
```

Figure 27: QUICStreamDataBlockedUpdated definition

5.19. datagram_data_blocked_updated

The datagram_data_blocked_updated event is used to indicate when QUIC datagrames becomes blocked or unblocked for sending. The event has Extra importance level.

```
QUICDatagramDataBlockedUpdated = {
    ? old: $BlockedState
    new: $BlockedState

    ? reason: $BlockedReason
}
```

Figure 28: QUICDatagramDataBlockedUpdated definition

5.20. migration_state_updated

Use to provide additional information when attempting (client-side) connection migration. While most details of the QUIC connection migration process can be inferred by observing the PATH_CHALLENGE and PATH_RESPONSE frames, in combination with the QUICPathAssigned event, it can be useful to explicitly log the progression of the migration and potentially made decisions in a single location/event. The event has Extra importance level.

Generally speaking, connection migration goes through two phases: a probing phase (which is not always needed/present), and a migration phase (which can be abandoned upon error).

Implementations that log per-path information in a QUICMigrationStateUpdated, SHOULD also emit QUICPathAssigned events, to serve as a ground-truth source of information.

Definition:

```
QUICMigrationStateUpdated = {
    ? old: MigrationState
    new: MigrationState

    ? path_id: PathID

    ; the information for traffic going towards the remote receiver
    ? path_remote: PathEndpointInfo

    ; the information for traffic coming in at the local endpoint
    ? path_local: PathEndpointInfo

    * $$quic-migrationstateupdated-extension
}

; Note that MigrationState does not describe a full state machine
; These entries are not necessarily chronological,
; nor will they always all appear during
; a connection migration attempt.
MigrationState =
    ; probing packets are sent, migration not initiated yet
    "probing_started" /
    ; did not get reply to probing packets,
    ; discarding path as an option
    "probing_abandoned" /
    ; received reply to probing packets, path is migration candidate
    "probing_successful" /
    ; non-probing packets are sent, attempting migration
    "migration_started" /
    ; something went wrong during the migration, abandoning attempt
    "migration_abandoned" /
    ; new path is now fully used, old path is discarded
    "migration_complete"
```

Figure 29: QUICMigrationStateUpdated definition

6. Security Events

6.1. key_updated

The key_updated event has Base importance level.

```
QUICKeyUpdated = {
    key_type: $KeyType
    ? old: hexstring
    ? new: hexstring

    ; needed for 1RTT key updates
    ? key_phase: uint64
    ? trigger:
        ; (e.g., initial, handshake and 0-RTT keys
        ; are generated by TLS)
        "tls" /
        "remote_update" /
        "local_update"

    * $$quic-keyupdated-extension
}
```

Figure 30: QUICKeyUpdated definition

Note that the key_phase is the full value of the key phase (as indicated by @M and @N in Figure 9 of [QUIC-TLS]). The key phase bit used on the packet header is the least significant bit of the key phase.

6.2. key_discarded

The key_discarded event has Base importance level.

```
QUICKeyDiscarded = {
    key_type: $KeyType
    ? key: hexstring

    ; needed for 1RTT key updates
    ? key_phase: uint64
    ? trigger:
        ; (e.g., initial, handshake and 0-RTT keys
        ; are generated by TLS)
        "tls" /
        "remote_update" /
        "local_update"

    * $$quic-keydiscarded-extension
}
```

Figure 31: QUICKeyDiscarded definition

7. Recovery events

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. recovery_parameters_set

The recovery_parameters_set event groups initial parameters from both loss detection and congestion control into a single event. It has Base importance level.

All these settings are typically set once and never change. Implementation that do, for some reason, change these parameters during execution, MAY emit the recovery_parameters_set event more than once.

```
QUICRecoveryParametersSet = {

    ; Loss detection, see RFC 9002 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, see RFC 9002 Appendix B.2
    ; in bytes. Note that this could be updated after pmtud
    ? max_datagram_size: uint32

    ; in bytes
    ? initial_congestion_window: uint64

    ; Note that 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

    * $$quic-recoveryparametersset-extension
}
```

Figure 32: QUICRecoveryParametersSet definition

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

7.2. recovery_metrics_updated

The recovery_metrics_updated event is emitted when one or more of the observable recovery metrics changes value. It has Core importance level.

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 recovery_metrics_updated entry, rather than split out into two). Consequently, a recovery_metrics_updated event is only guaranteed to contain at least one of the listed metrics.

```
QUICRecoveryMetricsUpdated = {

    ; Loss detection, see RFC 9002 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, see RFC 9002 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

    * $$quic-recoverymetricsupdated-extension
}
```

Figure 33: QUICRecoveryMetricsUpdated definition

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

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

7.3. congestion_state_updated

The congestion_state_updated event indicates when the congestion controller enters a significant new state and changes its behaviour. It has Base importance level.

The values of the event's fields are intentionally unspecified here in order to support different Congestion Control algorithms, as these typically have different states and even different implementations of these states across stacks. For example, for the algorithm defined in the QUIC Recovery RFC ("enhanced" New Reno), the following states are used: Slow Start, Congestion Avoidance, Application Limited and Recovery. Similarly, states can be triggered by a variety of events, including detection of Persistent Congestion or receipt of ECN markings.

```
QUICCongestionStateUpdated = {
    ? old: text
    new: text
    ? trigger: text

    * $$quic-congestionstateupdated-extension
}
```

Figure 34: QUICCongestionStateUpdated 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 (for example Slow Start is often exited only when ssthresh is exceeded).

7.4. loss_timer_updated

The loss_timer_updated event is emitted when a recovery loss timer changes state. It has Extra importance level.

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)

In order to indicate an active timer's timeout update, a new set event is used.

```
QUICLossTimerUpdated = {

    ; called "mode" in RFC 9002 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

    * $$quic-losstimerupdated-extension
}
```

Figure 35: QUICLossTimerUpdated definition

7.5. packet_lost

The packet_lost event is emitted when a packet is deemed lost by loss detection. It has Core importance level.

It is RECOMMENDED to populate the optional trigger field in order to help disambiguate among the various possible causes of a loss declaration.

```
QUICPacketLost = {

    ; 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" /
        ; RFC 9002 Section 6.2.4 paragraph 6, MAY
        "pto_expired"

    * $$quic-packetlost-extension
}
```

Figure 36: QUICPacketLost definition

7.6. marked_for_retransmit

The marked_for_retransmit event indicates which data was marked for retransmission upon detection of packet loss (see packet_lost). It has Extra importance level.

Similar to the 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.

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).

```
QUICMarkedForRetransmit = {
    frames: [+ $QuicFrame]

    * $$quic-markedforretransmit-extension
}
```

Figure 37: QUICMarkedForRetransmit definition

7.7. ecn_state_updated

The ecn_state_updated event indicates a progression in the ECN state machine as described in section A.4 of [QUIC-TRANSPORT]. It has Extra importance level.

```
QUICECNStateUpdated = {
   ? old: ECNState
    new: ECNState

    * $$quic-ecnstateupdated-extension
}

ECNState =
  ; ECN testing in progress
  "testing" /
  ; ECN state unknown, waiting for acknowledgements
  ; for testing packets
  "unknown" /
  ; ECN testing failed
  "failed" /
  ; testing was successful, the endpoint now
  ; sends packets with ECT(0) marking
  "capable"
```

Figure 38: QUICECNStateUpdated definition

8. QUIC data type definitions

8.1. QuicVersion

```
QuicVersion = hexstring
```

Figure 39: QuicVersion definition

8.2. ConnectionID

```
ConnectionID = hexstring
```

Figure 40: ConnectionID definition

8.3. Owner

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

Figure 41: Owner definition

8.4. IPAddress

```
; 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).
; Additionally, a hash-based or redacted representation
; can be used if needed for privacy or security reasons.
IPAddress = text /
            hexstring
```

Figure 42: IPAddress definition

8.5. PathEndpointInfo

PathEndpointInfo indicates a single half/direction of a path. A full path is comprised of two halves. Firstly: the server sends to the remote client IP + port using a specific destination Connection ID. Secondly: the client sends to the remote server IP + port using a different destination Connection ID.

As such, structures logging path information SHOULD include two different PathEndpointInfo instances, one for each half of the path.

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

    ; Even though usually only a single ConnectionID
    ; is associated with a given path at a time,
    ; there are situations where there can be an overlap
    ; or a need to keep track of previous ConnectionIDs
    ? connection_ids: [+ ConnectionID]

    * $$quic-pathendpointinfo-extension
}
```

Figure 43: PathEndpointInfo definition

8.6. PacketType

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

Figure 44: PacketType definition

8.7. PacketNumberSpace

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

Figure 45: PacketNumberSpace definition

8.8. PacketHeader

If the packet_type numerical value does not map to a known packet_type string, the packet_type value of "unknown" can be used and the raw value captured in the packet_type_bytes field; a numerical value without variable-length integer encoding.

```
PacketHeader = {
    ? quic_bit: bool .default true
    packet_type: $PacketType

    ; only if packet_type === "unknown"
    ? packet_type_bytes: uint64

    ; 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" || "retry"
    ? 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

    * $$quic-packetheader-extension
}
```

Figure 46: PacketHeader definition

8.9. Token

```
Token = {
    ? type: $TokenType

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

    * $$quic-token-extension
}

$TokenType /= "retry" /
              "resumption"
```

Figure 47: 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, Token includes a general-purpose details field.

8.10. Stateless Reset Token

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

Figure 48: Stateless Reset Token definition

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

8.11. 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 49: KeyType definition

8.12. ECN

```
ECN = "Not-ECT" / "ECT(1)" / "ECT(0)" / "CE"
```

Figure 50: ECN definition

The ECN bits carried in the IP header.

8.13. QUIC Frames

The generic $QuicFrame is defined here as a CDDL "type socket" extension point. 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 51: QuicFrame type socket 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 /
                  DatagramFrame

$QuicFrame /= QuicBaseFrames
```

Figure 52: QuicBaseFrames definition

8.13.1. 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 raw.payload_length property to the amount of PADDING bytes/frames included in the packet.

```
PaddingFrame = {
    frame_type: "padding"
    ? raw: RawInfo
}
```

Figure 53: PaddingFrame definition

8.13.2. PingFrame

```
PingFrame = {
    frame_type: "ping"
    ? raw: RawInfo
}
```

Figure 54: PingFrame definition

8.13.3. 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
    ? raw: RawInfo
}
```

Figure 55: AckFrame definition

Note that 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 that 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.

8.13.4. ResetStreamFrame

If the error_code numerical value does not map to a known ApplicationError string, the error_code value of "unknown" can be used and the raw value captured in the error_code_bytes field; a numerical value without variable-length integer encoding.

```
ResetStreamFrame = {
    frame_type: "reset_stream"
    stream_id: uint64
    error_code: $ApplicationError

    ; if error_code === "unknown"
    ? error_code_bytes: uint64

    ; in bytes
    final_size: uint64
    ? raw: RawInfo
}
```

Figure 56: ResetStreamFrame definition

8.13.5. StopSendingFrame

If the error_code numerical value does not map to a known ApplicationError string, the error_code value of "unknown" can be used and the raw value captured in the error_code_bytes field; a numerical value without variable-length integer encoding.

```
StopSendingFrame = {
    frame_type: "stop_sending"
    stream_id: uint64
    error_code: $ApplicationError

    ; if error_code === "unknown"
    ? error_code_bytes: uint64

    ? raw: RawInfo
}
```

Figure 57: StopSendingFrame definition

8.13.6. CryptoFrame

```
CryptoFrame = {
    frame_type: "crypto"
    offset: uint64
    length: uint64
    ? raw: RawInfo
}
```

Figure 58: CryptoFrame definition

8.13.7. NewTokenFrame

```
NewTokenFrame = {
  frame_type: "new_token"
  token: Token
  ? raw: RawInfo
}
```

Figure 59: NewTokenFrame definition

8.13.8. 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 60: StreamFrame definition

8.13.9. MaxDataFrame

```
MaxDataFrame = {
  frame_type: "max_data"
  maximum: uint64
  ? raw: RawInfo
}
```

Figure 61: MaxDataFrame definition

8.13.10. MaxStreamDataFrame

```
MaxStreamDataFrame = {
  frame_type: "max_stream_data"
  stream_id: uint64
  maximum: uint64
  ? raw: RawInfo
}
```

Figure 62: MaxStreamDataFrame definition

8.13.11. MaxStreamsFrame

```
MaxStreamsFrame = {
  frame_type: "max_streams"
  stream_type: StreamType
  maximum: uint64
  ? raw: RawInfo
}
```

Figure 63: MaxStreamsFrame definition

8.13.12. DataBlockedFrame

```
DataBlockedFrame = {
  frame_type: "data_blocked"
  limit: uint64
  ? raw: RawInfo
}
```

Figure 64: DataBlockedFrame definition

8.13.13. StreamDataBlockedFrame

```
StreamDataBlockedFrame = {
  frame_type: "stream_data_blocked"
  stream_id: uint64
  limit: uint64
  ? raw: RawInfo
}
```

Figure 65: StreamDataBlockedFrame definition

8.13.14. StreamsBlockedFrame

```
StreamsBlockedFrame = {
  frame_type: "streams_blocked"
  stream_type: StreamType
  limit: uint64
  ? raw: RawInfo
}
```

Figure 66: StreamsBlockedFrame definition

8.13.15. 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
  ? raw: RawInfo
}
```

Figure 67: NewConnectionIDFrame definition

8.13.16. RetireConnectionIDFrame

```
RetireConnectionIDFrame = {
  frame_type: "retire_connection_id"
  sequence_number: uint32
  ? raw: RawInfo
}
```

Figure 68: RetireConnectionIDFrame definition

8.13.17. PathChallengeFrame

```
PathChallengeFrame = {
  frame_type: "path_challenge"

  ; always 64 bits
  ? data: hexstring
  ? raw: RawInfo
}
```

Figure 69: PathChallengeFrame definition

8.13.18. PathResponseFrame

```
PathResponseFrame = {
  frame_type: "path_response"

  ; always 64 bits
  ? data: hexstring
  ? raw: RawInfo
}
```

Figure 70: PathResponseFrame definition

8.13.19. ConnectionCloseFrame

An endpoint that receives unknown error codes can record it in the error_code field using the numerical value without variable-length integer encoding.

When the connection is closed due a connection-level error, the trigger_frame_type field can be used to log the frame that triggered the error. For known frame types, the appropriate string value is used in error_code. For unknown frame types, the error_code field has the value "unknown" and the numerical value without variable-length integer encoding is logged in error_code_bytes.

The CONNECTION_CLOSE reason phrase is a byte sequences. It is likely that this sequence is presentable as UTF-8, in which case it can be logged in the reason field. The reason_bytes field supports logging the raw bytes, which can be useful when the value is not UTF-8 or when an endpoint does not want to decode it. Implementations SHOULD log at least one format, but MAY log both or none.

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

ConnectionCloseFrame = {
    frame_type: "connection_close"
    ? error_space: ErrorSpace
    ? error_code: $TransportError /
                  CryptoError /
                  $ApplicationError

    ; only if error_code === "unknown"
    ? error_code_bytes: uint64

    ? reason: text
    ? reason_bytes: hexstring

    ; when error_space === "transport"
    ? trigger_frame_type: uint64 /
                          text
    ? raw: RawInfo
}
```

Figure 71: ConnectionCloseFrame definition

8.13.20. HandshakeDoneFrame

```
HandshakeDoneFrame = {
  frame_type: "handshake_done"
  ? raw: RawInfo
}
```

Figure 72: HandshakeDoneFrame definition

8.13.21. UnknownFrame

The frame_type_bytes field is the numerical value without variable-length integer encoding.

```
UnknownFrame = {
    frame_type: "unknown"
    frame_type_bytes: uint64
    ? raw: RawInfo
}
```

Figure 73: UnknownFrame definition

8.13.22. DatagramFrame

The QUIC DATAGRAM frame is defined in Section 4 of [RFC9221].

```
DatagramFrame = {
    frame_type: "datagram"
    ? length: uint64
    ? raw: RawInfo
}
```

Figure 74: DatagramFrame definition

8.13.23. TransportError

The generic $TransportError is defined here as a CDDL "type socket" extension point. It can be extended to support additional Transport errors.

```
$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" /
                   "unknown"
                   ; there is no value to reflect CRYPTO_ERROR
                   ; use the CryptoError type instead
```

Figure 75: TransportError definition

8.13.24. 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 completely. It is instead defined as a CDDL "type socket" extension point, with a single "unknown" value.

```
$ApplicationError /= "unknown"
```

Figure 76: ApplicationError definition

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"
```

8.13.25. 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 77: 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

This document registers a new entry in the "qlog event schema URIs" registry (created in Section 15 of [QLOG-MAIN]):

Event schema URI: urn:ietf:params:qlog:events:quic

Namespace quic

Event Types server_listening,connection_started,connection_closed,connection_id_updated,spin_bit_updated,connection_state_updated,path_assigned,mtu_updated,version_information,alpn_information,parameters_set,parameters_restored,packet_sent,packet_received,packet_dropped,packet_buffered,packets_acked,udp_datagrams_sent,udp_datagrams_received,udp_datagram_dropped,stream_state_updated,frames_processed,stream_data_moved,datagram_data_moved,migration_state_updated,key_updated,key_discarded,recovery_parameters_set,recovery_metrics_updated,congestion_state_updated,loss_timer_updated,packet_lost,marked_for_retransmit,ecn_state_updated

Description: Event definitions related to the QUIC transport protocol.

Reference: This Document

11. References

11.1. 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>.

[GREASEBIT] Thomson, M., "Greasing the QUIC Bit", RFC 9287, DOI 10.17487/RFC9287, August 2022, <https://www.rfc-editor.org/rfc/rfc9287>.

[QLOG-MAIN] Marx, R., Niccolini, L., Seemann, M., and L. Pardue, "qlog: Structured Logging for Network Protocols", Work in Progress, Internet-Draft, draft-ietf-quic-qlog-main-schema-11, 17 March 2025, <https://datatracker.ietf.org/doc/html/draft-ietf-quic-qlog-main-schema-11>.

[QUIC-DATAGRAM] Pauly, T., Kinnear, E., and D. Schinazi, "An Unreliable Datagram Extension to QUIC", RFC 9221, DOI 10.17487/RFC9221, March 2022, <https://www.rfc-editor.org/rfc/rfc9221>.

[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] Thomson, M., Ed. and S. Turner, Ed., "Using TLS to Secure QUIC", RFC 9001, DOI 10.17487/RFC9001, May 2021, <https://www.rfc-editor.org/rfc/rfc9001>.

[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>.

[RFC9221] Pauly, T., Kinnear, E., and D. Schinazi, "An Unreliable Datagram Extension to QUIC", RFC 9221, DOI 10.17487/RFC9221, March 2022, <https://www.rfc-editor.org/rfc/rfc9221>.

11.2. Informative References

[RFC7301] Friedl, S., Popov, A., Langley, A., and E. Stephan, "Transport Layer Security (TLS) Application-Layer Protocol Negotiation Extension", RFC 7301, DOI 10.17487/RFC7301, July 2014, <https://www.rfc-editor.org/rfc/rfc7301>.

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, Christian Huitema, Hugo Landau, Will Hawkins, Mathis Engelbart and Jonathan Lennox for their feedback and suggestions.

Change Log

This section is to be removed before publishing as an RFC.

Since draft-ietf-qlog-quic-events-09:

* Several editorial changes

* Reworked QUICConnectionStarted to use PathEndpointInfo (#453)

* Consistent use of RawInfo and _bytes fields to log raw data (#450)

Since draft-ietf-qlog-quic-events-08:

* Removed individual categories and put every event in the single quic event schema namespace. Major change (#439)

* Renamed recovery:metrics_updated to quic:recovery_metrics_updated and recovery:parameters_set to quic:recovery_parameters_set (#439)

* Added unknown_parameters field to parameters_set (#438)

* Added extra parameters to parameters_restored (#441)

Since draft-ietf-qlog-quic-events-07:

* TODO (we forgot...)

Since draft-ietf-qlog-quic-events-06:

* Added PathAssigned and MigrationStateUpdated events (#336)

* Added extension points to parameters_set and parameters_restored (#400)

* Removed error_code_value from connection_closed (#386, #392)

* Renamed generation to key_phase for key_updated and key_discarded (#390)

* Removed retry_token from packet_sent and packet_received (#389)

* Updated ALPN handling (#385)

* Added key_unavailable trigger to packet_dropped (#381)

* Updated several uint32 to uint64

* ProtocolEventBody is now called ProtocolEventData (#352)

* Editorial changes (#402, #404, #394, #393)

Since draft-ietf-qlog-quic-events-05:

* SecurityKeyUpdated: the new key is no longer mandatory to log (#294)

* Added ECN related events and metadata (#263)

Since draft-ietf-qlog-quic-events-04:

* Updated guidance on logging events across connections (#279)

* Renamed 'transport' category to 'quic' (#302)

* Added support for multiple packet numbers in 'quic:frames_processed' (#307)

* Added definitions for RFC9287 (QUIC GREASE Bit extension) (#311)

* Added definitions for RFC9221 (QUIC Datagram Frame extension) (#310)

* (Temporarily) removed definitions for connection migration events (#317)

* Editorial and formatting changes (#298, #299, #304, #306, #327)

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)

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

Since draft-ietf-qlog-quic-events-01:

* Added Stateless Reset Token type (#122)

Since draft-ietf-qlog-quic-events-00:

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

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

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)

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)

Authors' Addresses

Robin Marx (editor)
Akamai
Email: rmarx@akamai.com

Luca Niccolini (editor)
Meta
Email: lniccolini@meta.com

Marten Seemann (editor)
Email: martenseemann@gmail.com

Lucas Pardue (editor)
Cloudflare
Email: lucas@lucaspardue.com