

QUIC  
Internet-Draft  
Intended status: Standards Track  
Expires: 6 May 2021

R. Marx  
Hasselt University  
2 November 2020

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

## Abstract

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

## Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at <https://datatracker.ietf.org/drafts/current/>.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on 6 May 2021.

## Copyright Notice

Copyright (c) 2020 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<https://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

## Table of Contents

1. Introduction . . . . .	5
1.1. Notational Conventions . . . . .	5
2. Overview . . . . .	5
2.1. Importance . . . . .	6
2.2. Custom fields . . . . .	7
3. Events not belonging to a single connection . . . . .	7
4. QUIC and HTTP/3 fields . . . . .	8
4.1. Raw packet and frame information . . . . .	8
5. QUIC event definitions . . . . .	10
5.1. connectivity . . . . .	10
5.1.1. server_listening . . . . .	10
5.1.2. connection_started . . . . .	10
5.1.3. connection_closed . . . . .	11
5.1.4. connection_id_updated . . . . .	12
5.1.5. spin_bit_updated . . . . .	12
5.1.6. connection_retried . . . . .	12
5.1.7. connection_state_updated . . . . .	13
5.1.8. MIGRATION-related events . . . . .	15
5.2. security . . . . .	15
5.2.1. key_updated . . . . .	15
5.2.2. key_retired . . . . .	15
5.3. transport . . . . .	16
5.3.1. version_information . . . . .	16
5.3.2. alpn_information . . . . .	17
5.3.3. parameters_set . . . . .	18
5.3.4. parameters_restored . . . . .	20
5.3.5. packet_sent . . . . .	20
5.3.6. packet_received . . . . .	21
5.3.7. packet_dropped . . . . .	22
5.3.8. packet_buffered . . . . .	23
5.3.9. packets_acked . . . . .	24
5.3.10. datagrams_sent . . . . .	24
5.3.11. datagrams_received . . . . .	25
5.3.12. datagram_dropped . . . . .	25
5.3.13. stream_state_updated . . . . .	26
5.3.14. frames_processed . . . . .	27
5.3.15. data_moved . . . . .	28
5.4. recovery . . . . .	30
5.4.1. parameters_set . . . . .	30
5.4.2. metrics_updated . . . . .	30
5.4.3. congestion_state_updated . . . . .	31
5.4.4. loss_timer_updated . . . . .	32
5.4.5. packet_lost . . . . .	33
5.4.6. marked_for_retransmit . . . . .	34
6. HTTP/3 event definitions . . . . .	34
6.1. http . . . . .	34

6.1.1.	parameters_set . . . . .	34
6.1.2.	parameters_restored . . . . .	35
6.1.3.	stream_type_set . . . . .	36
6.1.4.	frame_created . . . . .	36
6.1.5.	frame_parsed . . . . .	37
6.1.6.	push_resolved . . . . .	37
6.2.	qpack . . . . .	38
6.2.1.	state_updated . . . . .	38
6.2.2.	stream_state_updated . . . . .	39
6.2.3.	dynamic_table_updated . . . . .	39
6.2.4.	headers_encoded . . . . .	39
6.2.5.	headers_decoded . . . . .	40
6.2.6.	instruction_created . . . . .	40
6.2.7.	instruction_parsed . . . . .	41
7.	Generic events and Simulation indicators . . . . .	41
7.1.	generic . . . . .	41
7.1.1.	error . . . . .	42
7.1.2.	warning . . . . .	42
7.1.3.	info . . . . .	42
7.1.4.	debug . . . . .	42
7.1.5.	verbose . . . . .	43
7.2.	simulation . . . . .	43
7.2.1.	scenario . . . . .	43
7.2.2.	marker . . . . .	44
8.	Security Considerations . . . . .	44
9.	IANA Considerations . . . . .	44
10.	References . . . . .	44
10.1.	Normative References . . . . .	44
10.2.	Informative References . . . . .	45
Appendix A.	QUIC data field definitions . . . . .	45
A.1.	IPAddress . . . . .	45
A.2.	PacketType . . . . .	45
A.3.	PacketNumberSpace . . . . .	45
A.4.	PacketHeader . . . . .	45
A.5.	Token . . . . .	46
A.6.	KeyType . . . . .	46
A.7.	QUIC Frames . . . . .	47
A.7.1.	PaddingFrame . . . . .	47
A.7.2.	PingFrame . . . . .	47
A.7.3.	AckFrame . . . . .	47
A.7.4.	ResetStreamFrame . . . . .	48
A.7.5.	StopSendingFrame . . . . .	48
A.7.6.	CryptoFrame . . . . .	49
A.7.7.	NewTokenFrame . . . . .	49
A.7.8.	StreamFrame . . . . .	49
A.7.9.	MaxDataFrame . . . . .	50
A.7.10.	MaxStreamDataFrame . . . . .	50
A.7.11.	MaxStreamsFrame . . . . .	50

A.7.12.	DataBlockedFrame . . . . .	50
A.7.13.	StreamDataBlockedFrame . . . . .	50
A.7.14.	StreamsBlockedFrame . . . . .	50
A.7.15.	NewConnectionIDFrame . . . . .	51
A.7.16.	RetireConnectionIDFrame . . . . .	51
A.7.17.	PathChallengeFrame . . . . .	51
A.7.18.	PathResponseFrame . . . . .	51
A.7.19.	ConnectionCloseFrame . . . . .	52
A.7.20.	HandshakeDoneFrame . . . . .	52
A.7.21.	UnknownFrame . . . . .	52
A.7.22.	TransportError . . . . .	52
A.7.23.	CryptoError . . . . .	53
Appendix B.	HTTP/3 data field definitions . . . . .	53
B.1.	HTTP/3 Frames . . . . .	53
B.1.1.	DataFrame . . . . .	53
B.1.2.	HeadersFrame . . . . .	54
B.1.3.	CancelPushFrame . . . . .	54
B.1.4.	SettingsFrame . . . . .	54
B.1.5.	PushPromiseFrame . . . . .	54
B.1.6.	GoAwayFrame . . . . .	55
B.1.7.	MaxPushIDFrame . . . . .	55
B.1.8.	DuplicatePushFrame . . . . .	55
B.1.9.	ReservedFrame . . . . .	55
B.1.10.	UnknownFrame . . . . .	55
B.2.	ApplicationError . . . . .	55
Appendix C.	QPACK DATA type definitions . . . . .	56
C.1.	QPACK Instructions . . . . .	56
C.1.1.	SetDynamicTableCapacityInstruction . . . . .	56
C.1.2.	InsertWithNameReferenceInstruction . . . . .	56
C.1.3.	InsertWithoutNameReferenceInstruction . . . . .	57
C.1.4.	DuplicateInstruction . . . . .	57
C.1.5.	HeaderAcknowledgementInstruction . . . . .	57
C.1.6.	StreamCancellationInstruction . . . . .	57
C.1.7.	InsertCountIncrementInstruction . . . . .	58
C.2.	QPACK Header compression . . . . .	58
C.2.1.	IndexedHeaderField . . . . .	58
C.2.2.	LiteralHeaderFieldWithName . . . . .	58
C.2.3.	LiteralHeaderFieldWithoutName . . . . .	59
C.2.4.	QPackHeaderBlockPrefix . . . . .	59
Appendix D.	Change Log . . . . .	59
D.1.	Since draft-01: . . . . .	59
D.2.	Since draft-00: . . . . .	61
Appendix E.	Design Variations . . . . .	61
Appendix F.	Acknowledgements . . . . .	61
Author's Address	. . . . .	61

## 1. Introduction

This document describes the values of the qlog name ("category" + "event") and "data" fields and their semantics for the QUIC and HTTP/3 protocols. This document is based on draft-29 of the QUIC and HTTP/3 I-Ds QUIC-TRANSPORT [QUIC-HTTP] and draft-16 of the QPACK I-D [QUIC-QPACK].

Feedback and discussion welcome at <https://github.com/quiclog/internet-drafts> (<https://github.com/quiclog/internet-drafts>). Readers are advised to refer to the "editor's draft" at that URL for an up-to-date version of this document.

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

### 1.1. Notational Conventions

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

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

## 2. Overview

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

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

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

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

## 2.1. Importance

Many of the events defined in this document map directly to concepts seen in the QUIC and HTTP/3 documents, while others act as aggregating events that combine data from several possible protocol behaviours or code paths into one. This is done to reduce the amount of unique event definitions, as reflecting each possible protocol event as a separate qlog entity would cause an explosion of event types. Similarly, we prevent logging duplicate packet data as much as possible. As such, especially packet header value updates are split out into separate events (for example `spin_bit_updated`, `connection_id_updated`), as they are expected to change sparingly.

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

To aid in this decision making, each event has an "importance indicator" with one of three values, in decreasing order of importance and expected usage:

- \* Core
- \* Base
- \* Extra

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

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

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

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

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

## 2.2. Custom fields

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

## 3. Events not belonging to a single connection

For several types of events, it is sometimes impossible to tie them to a specific conceptual QUIC connection (e.g., a packet\_dropped event triggered because the packet has an unknown connection\_id in the header). Since qlog events in a trace are typically associated with a single connection, it is unclear how to log these events.

Ideally, implementers SHOULD create a separate, individual "endpoint-level" trace file (or group\_id value), not associated with a specific connection (for example a "server.qlog" or group\_id = "client"), and log all events that do not belong to a single connection to this grouping trace. However, this is not always practical, depending on the implementation. Because the semantics of most of these events are well-defined in the protocols and because they are difficult to mis-interpret as belonging to a connection, implementers MAY choose to log events not belonging to a particular connection in any other trace, even those strongly associated with a single connection.

Note that this can make it difficult to match logs from different vantage points with each other. For example, from the client side, it is easy to log connections with version negotiation or retry in the same trace, while on the server they would most likely be logged in separate traces. Servers can take extra efforts (and keep additional state) to keep these events combined in a single trace however (for example by also matching connections on their four-tuple instead of just the connection ID).

#### 4. QUIC and HTTP/3 fields

This document re-uses all the fields defined in the main qlog schema (e.g., name, category, type, data, group\_id, protocol\_type, the time-related fields, etc.).

The value of the "protocol\_type" qlog field MUST be "QUIC\_HTTP3".

When the qlog "group\_id" field is used, it is recommended to use QUIC's Original Destination Connection ID (ODCID, the CID chosen by the client when first contacting the server), as this is the only value that does not change over the course of the connection and can be used to link more advanced QUIC packets (e.g., Retry, Version Negotiation) to a given connection. Similarly, the ODCID should be used as the qlog filename or file identifier, potentially suffixed by the vantagepoint type (For example, abcd1234\_server.qlog would contain the server-side trace of the connection with ODCID abcd1234).

##### 4.1. Raw packet and frame information

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



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

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

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

For QUIC packets:  $\text{header\_length} = \text{length} - \text{payload\_length} - \text{aead\_tag\_length}$

For QUIC and HTTP/3 frames:  $\text{header\_length} = \text{length} - \text{payload\_length}$

For UDP datagrams:  $\text{header\_length} = \text{length} - \text{payload\_length}$

Note: In some cases, the length fields are also explicitly reflected inside of frame/packet headers. For example, the QUIC STREAM frame has a "length" field indicating its payload size. Similarly, all HTTP/3 frames include their explicit payload lengths in the frame header. Finally, the QUIC Long Header has a "length" field which is equal to the payload length plus the packet number length. In these cases, those fields are intentionally preserved in the event definitions. Even though this can lead to duplicate data when the full RawInfo is logged, it allows a more direct mapping of the QUIC and HTTP/3 specifications to qlog, making it easier for users to interpret.

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

## 5. QUIC event definitions

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

### 5.1. connectivity

#### 5.1.1. server\_listening

Importance: Extra

Emitted when the server starts accepting connections.

Data:

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

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

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

#### 5.1.2. connection\_started

Importance: Base

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

Data:

```

{
  ip_version?: "v4" | "v6",
  src_ip?: IPAddress,
  dst_ip?: IPAddress,

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

  src_cid?: bytes,
  dst_cid?: bytes,
}

```

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

#### 5.1.3. connection\_closed

Importance: Base

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

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

```

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

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

  reason?: string
}

```

Triggers: \* clean \* handshake\_timeout \* idle\_timeout \* error // this is called the "immediate close" in the QUIC specification \* stateless\_reset \* version\_mismatch \* application // for example HTTP/3's GOAWAY frame

#### 5.1.4. connection\_id\_updated

Importance: Base

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

This is viewed from the perspective of the one applying the new id. As such, if we receive a new connection id from our peer, we will see the dst\_ fields are set. If we update our own connection id (e.g., NEW\_CONNECTION\_ID frame), we log the src\_ fields.

Data:

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

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

#### 5.1.5. spin\_bit\_updated

Importance: Base

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

Data:

```
{
  state: boolean
}
```

#### 5.1.6. connection\_retried

TODO

## 5.1.7. connection\_state\_updated

Importance: Base

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

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

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

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

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

\*Client:\*

\* send initial

- state = attempted

\* get initial

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

\* get first Handshake packet

- state = handshake\_started

- \* get Handshake packet containing ServerFinished
  - state = handshake\_complete
- \* send ClientFinished
  - state = early\_write (1RTT can now be sent)
- \* get HANDSHAKE\_DONE
  - state = handshake\_confirmed
- \*Server:\*
- \* get initial
  - state = attempted
- \* send initial \_(don't think this needs a separate state, since some handshake will always be sent in the same flight as this?)\_
- \* send handshake EE, CERT, CV, ...
  - state = handshake\_started
- \* send ServerFinished
  - state = early\_write (1RTT can now be sent)
- \* get first handshake packet / something using a server-issued CID of min length
  - state = validated
- \* get handshake packet containing ClientFinished
  - state = handshake\_complete
- \* send HANDSHAKE\_DONE
  - state = handshake\_confirmed

Note: connection\_state\_changed with a new state of "attempted" is the same conceptual event as the connection\_started event above from the client's perspective. Similarly, a state of "closing" or "draining" corresponds to the connection\_closed event.

#### 5.1.8. MIGRATION-related events

e.g., path\_updated

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

For now, infer from other connectivity events and path\_challenge/  
path\_response frames

#### 5.2. security

##### 5.2.1. key\_updated

Importance: Base

Note: secret\_updated would be more correct, but in the draft it's called KEY\_UPDATE, so stick with that for consistency

Data:

```
{
  key_type:KeyType,
  old?:bytes,
  new:bytes,
  generation?:uint32 // needed for 1RTT key updates
}
```

Triggers:

- \* "tls" // (e.g., initial, handshake and 0-RTT keys are generated by TLS)
- \* "remote\_update"
- \* "local\_update"

##### 5.2.2. key\_retired

Importance: Base

Data:

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

Triggers:

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

### 5.3. transport

#### 5.3.1. version\_information

Importance: Core

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

Data:

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

Intended use:

- \* When sending an initial, the client logs this event with client\_versions and chosen\_version set
- \* Upon receiving a client initial with a supported version, the server logs this event with server\_versions and chosen\_version set



- \* Upon receiving a client initial with an unsupported version, the server logs this event with `server_versions` set and `client_versions` to the single-element array containing the client's attempted version. The absence of `chosen_version` implies no overlap was found.
- \* Upon receiving a version negotiation packet from the server, the client logs this event with `client_versions` set and `server_versions` to the versions in the version negotiation packet and `chosen_version` to the version it will use for the next initial packet

#### 5.3.2. `alpn_information`

Importance: Core

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

Data:

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

Intended use:

- \* When sending an initial, the client logs this event with `client_alpns` set
- \* When receiving an initial with a supported alpn, the server logs this event with `server_alpns` set, `client_alpns` equalling the client-provided list, and `chosen_alpn` to the value it will send back to the client.
- \* When receiving an initial with an alpn, the client logs this event with `chosen_alpn` to the received value.
- \* Alternatively, a client can choose to not log the first event, but wait for the receipt of the server initial to log this event with both `client_alpns` and `chosen_alpn` set.

### 5.3.3. parameters\_set

Importance: Core

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

All these settings are typically set once and never change. However, they are typically set at different times during the connection, so there will typically be several instances of this event with different fields set.

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

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

Data:

```

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

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

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

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

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

    preferred_address?: PreferredAddress
}

interface PreferredAddress {
    ip_v4: IPAddress,
    ip_v6: IPAddress,

    port_v4: uint16,
    port_v6: uint16,

    connection_id: bytes,
    stateless_reset_token: Token
}

```

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

#### 5.3.4. parameters\_restored

Importance: Base

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

Data:

```
{
  disable_active_migration?:boolean,

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

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

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

#### 5.3.5. packet\_sent

Importance: Core

Data:

```
{
  header:PacketHeader,

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

  is_coalesced?:boolean, // default value is false

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

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

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

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

Note: We do not explicitly log the encryption\_level or packet\_number\_space: the header.packet\_type specifies this by inference (assuming correct implementation)

Triggers:

- \* "retransmit\_reordered" // draft-23 5.1.1
- \* "retransmit\_timeout" // draft-23 5.1.2
- \* "pto\_probe" // draft-23 5.3.1
- \* "retransmit\_crypto" // draft-19 6.2
- \* "cc\_bandwidth\_probe" // needed for some CCs to figure out bandwidth allocations when there are no normal sends

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

#### 5.3.6. packet\_received

Importance: Core

Data:

```
{
  header:PacketHeader,

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

  is_coalesced?:boolean,

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

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

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

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

Note: We do not explicitly log the encryption\_level or packet\_number\_space: the header.packet\_type specifies this by inference (assuming correct implementation)

Triggers:

\* "keys\_available" // if packet was buffered because it couldn't be decrypted before

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

### 5.3.7. packet\_dropped

Importance: Base

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

Data:

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

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

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

Triggers:

- \* "key\_unavailable"
- \* "unknown\_connection\_id"
- \* "header\_parse\_error"
- \* "payload\_decrypt\_error"
- \* "protocol\_violation"
- \* "dos\_prevention"
- \* "unsupported\_version"
- \* "unexpected\_packet"
- \* "unexpected\_source\_connection\_id"
- \* "unexpected\_version"
- \* "duplicate"
- \* "invalid\_initial"

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

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

#### 5.3.8. packet\_buffered

Importance: Base

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

Data:

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

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

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

Triggers:

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

#### 5.3.9. packets\_acked

Importance: Extra

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

Data: ~~~ { packet\_number\_space?:PacketNumberSpace,  
packet\_numbers?:Array<uint64> } ~~~

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

#### 5.3.10. datagrams\_sent

Importance: Extra

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

Data:



```

{
    count?:uint16, // to support passing multiple at once
    raw?:Array<RawInfo>, // RawInfo:length field indicates total length of the da
tagrams, including UDP header length

    datagram_ids?:Array<uint32>
}

```

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

#### 5.3.11. datagrams\_received

Importance: Extra

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

Data:

```

{
    count?:uint16, // to support passing multiple at once
    raw?:Array<RawInfo>, // RawInfo:length field indicates total length of the da
tagrams, including UDP header length

    datagram_ids?:Array<uint32>
}

```

Note: for more details on "datagram\_ids", see Section 5.3.10.

#### 5.3.12. datagram\_dropped

Importance: Extra

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

Data:

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

#### 5.3.13. stream\_state\_updated

Importance: Base

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

Data:

```

{
    stream_id:uint64,
    stream_type?:"unidirectional"|"bidirectional", // mainly useful when opening
the stream

    old?:StreamState,
    new:StreamState,

    stream_side?:"sending"|"receiving"
}

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

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

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

    // both-side states
    data_received,

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

```

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

#### 5.3.14. frames\_processed

Importance: Extra

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

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

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

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

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

Data:

```
{
  frames:Array<QuicFrame>, // see appendix for the definitions
  packet_number?:uint64
}
```

#### 5.3.15. `data_moved`

Importance: Base

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

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

Data:

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

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

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

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

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

#### 5.4. recovery

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

##### 5.4.1. parameters\_set

Importance: Base

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

Data:

```
{
  // Loss detection, see recovery draft-23, Appendix A.2
  reordering_threshold?:uint16, // in amount of packets
  time_threshold?:float, // as RTT multiplier
  timer_granularity?:uint16, // in ms
  initial_rtt?:float, // in ms

  // congestion control, Appendix B.1.
  max_datagram_size?:uint32, // in bytes // Note: this could be updated after p
mtud
  initial_congestion_window?:uint64, // in bytes
  minimum_congestion_window?:uint32, // in bytes // Note: this could change whe
n max_datagram_size changes
  loss_reduction_factor?:float,
  persistent_congestion_threshold?:uint16 // as PTO multiplier
}
```

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

##### 5.4.2. metrics\_updated

Importance: Core

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

Data:

```
{
    // Loss detection, see recovery draft-23, Appendix A.3
    min_rtt?:float, // in ms or us, depending on the overarching qlog's configura
tion
    smoothed_rtt?:float, // in ms or us, depending on the overarching qlog's conf
iguration
    latest_rtt?:float, // in ms or us, depending on the overarching qlog's config
uration
    rtt_variance?:float, // in ms or us, depending on the overarching qlog's conf
iguration

    pto_count?:uint16,

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

    ssthresh?:uint64, // in bytes

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

    pacing_rate?:uint64 // in bps
}
```

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

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

#### 5.4.3. `congestion_state_updated`

Importance: Base

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

- \* slow\_start
- \* congestion\_avoidance
- \* application\_limited
- \* recovery

Data:

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

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

Some triggers for ("enhanced" New Reno):

- \* persistent\_congestion
- \* ECN

#### 5.4.4. loss\_timer\_updated

Importance: Extra

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

- \* set: the timer is set with a delta timeout for when it will trigger next
- \* expired: when the timer effectively expires after the delta timeout
- \* cancelled: when a timer is cancelled (e.g., all outstanding packets are acknowledged, start idle period)



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

Data:

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

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

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

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

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

#### 5.4.5. packet\_lost

Importance: Core

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

Data:

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

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

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

Triggers:

- \* "reordering\_threshold",
- \* "time\_threshold"
- \* "pto\_expired" // draft-23 section 5.3.1, MAY

#### 5.4.6. marked\_for\_retransmit

Importance: Extra

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

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

Data:

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

### 6. HTTP/3 event definitions

#### 6.1. http

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

##### 6.1.1. parameters\_set

Importance: Base

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

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

Data:

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

  max_header_list_size?: uint64, // from SETTINGS_MAX_HEADER_LIST_SIZE
  max_table_capacity?: uint64, // from SETTINGS_QPACK_MAX_TABLE_CAPACITY
  blocked_streams_count?: uint64, // from SETTINGS_QPACK_BLOCKED_STREAMS

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

Note: enabling server push is not explicitly done in HTTP/3 by use of a setting or parameter. Instead, it is communicated by use of the MAX\_PUSH\_ID frame, which should be logged using the frame\_created and frame\_parsed events below.

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

#### 6.1.2. parameters\_restored

Importance: Base

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

Data:

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

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

### 6.1.3. stream\_type\_set

Importance: Base

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

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

Data:

```
{
  stream_id:uint64,

  owner?:"local"|"remote"

  old?:StreamType,
  new:StreamType,

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

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

### 6.1.4. frame\_created

Importance: Core

HTTP equivalent to the packet\_sent event. This event is emitted when the HTTP/3 framing actually happens. Note: this is not necessarily the same as when the HTTP/3 data is passed on to the QUIC layer. For that, see the "data\_moved" event.

Data:

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

  raw?:RawInfo
}
```

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

#### 6.1.5. `frame_parsed`

Importance: Core

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

Data:

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

  raw?:RawInfo
}
```

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

#### 6.1.6. `push_resolved`

Importance: Extra

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

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

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

## 6.2. qpack

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

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

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

### 6.2.1. state\_updated

Importance: Base

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

Data:

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

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

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

#### 6.2.2. stream\_state\_updated

Importance: Core

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

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

Data:

```
{
  stream_id:uint64,

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

#### 6.2.3. dynamic\_table\_updated

Importance: Extra

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

Data:

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

  update_type:"inserted"|"evicted",

  entries:Array<DynamicTableEntry>
}

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

#### 6.2.4. headers\_encoded

Importance: Base

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

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

Data:

```
{
  stream_id?:uint64,

  headers?:Array<HTTPHeader>,

  block_prefix:QPackHeaderBlockPrefix,
  header_block:Array<QPackHeaderBlockRepresentation>,

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

#### 6.2.5. headers\_decoded

Importance: Base

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

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

Data:

```
{
  stream_id?:uint64,

  headers?:Array<HTTPHeader>,

  block_prefix:QPackHeaderBlockPrefix,
  header_block:Array<QPackHeaderBlockRepresentation>,

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

#### 6.2.6. instruction\_created

Importance: Base



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

Data:

```
{
  instruction:QPackInstruction // see appendix for the definitions,
  length?:uint32,
  raw?:bytes
}
```

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

#### 6.2.7. instruction\_parsed

Importance: Base

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

Data:

```
{
  instruction:QPackInstruction // see appendix for the definitions,
  length?:uint32,
  raw?:bytes
}
```

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

### 7. Generic events and Simulation indicators

#### 7.1. generic

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

#### 7.1.1. error

Importance: Core

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

Data:

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

#### 7.1.2. warning

Importance: Base

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

Data:

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

#### 7.1.3. info

Importance: Extra

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

Data:

```
{
  message:string
}
```

#### 7.1.4. debug

Importance: Extra

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

Data:

```
{
  message:string
}
```

#### 7.1.5. verbose

Importance: Extra

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

Data:

```
{
  message:string
}
```

#### 7.2. simulation

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

##### 7.2.1. scenario

Importance: Extra

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

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

### 7.2.2. marker

Importance: Extra

Used to indicate when specific emulation conditions are triggered at set times (e.g., at 3 seconds in 2% packet loss is introduced, at 10s a NAT rebind is triggered).

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

## 8. Security Considerations

TBD

## 9. IANA Considerations

TBD

## 10. References

### 10.1. Normative References

#### [QLOG-MAIN]

Marx, R., Ed., "Main logging schema for qlog", Work in Progress, Internet-Draft, draft-marx-qlog-main-schema-02, 2 November 2020, <<https://tools.ietf.org/html/draft-marx-qlog-main-schema-02>>.

#### [QUIC-HTTP]

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

#### [QUIC-QPACK]

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

[QUIC-TRANSPORT]

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

10.2. Informative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <<https://www.rfc-editor.org/info/rfc2119>>.

Appendix A. QUIC data field definitions

A.1. IPAddress

```
class IPAddress : string | bytes;
```

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

A.2. PacketType

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

A.3. PacketNumberSpace

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

A.4. PacketHeader

```

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

    packet_type: PacketType;
    packet_number: uint64;

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

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

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

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

```

#### A.5. Token

```

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

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

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

```

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

#### A.6. KeyType

```
enum KeyType {
    server_initial_secret,
    client_initial_secret,

    server_handshake_secret,
    client_handshake_secret,

    server_0rtt_secret,
    client_0rtt_secret,

    server_1rtt_secret,
    client_1rtt_secret
}
```

#### A.7. QUIC Frames

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

##### A.7.1. PaddingFrame

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

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

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

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

##### A.7.2. PingFrame

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

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

##### A.7.3. AckFrame

```

class AckFrame{
    frame_type:string = "ack";

    ack_delay?:float; // in ms

    // first number is "from": lowest packet number in interval
    // second number is "to": up to and including // highest packet number in interval
    // e.g., looks like [[1,2],[4,5]]
    acked_ranges?:Array<[uint64, uint64]|[uint64]>;

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

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

```

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

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

#### A.7.4. ResetStreamFrame

```

class ResetStreamFrame{
    frame_type:string = "reset_stream";

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

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

```

#### A.7.5. StopSendingFrame



```

class StopSendingFrame{
    frame_type:string = "stop_sending";

    stream_id:uint64;
    error_code:ApplicationError | uint32;

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

```

## A.7.6. CryptoFrame

```

class CryptoFrame{
    frame_type:string = "crypto";

    offset:uint64;
    length:uint64;

    payload_length?:uint32;
}

```

## A.7.7. NewTokenFrame

```

class NewTokenFrame{
    frame_type:string = "new_token";

    token:Token
}

```

## A.7.8. StreamFrame

```

class StreamFrame{
    frame_type:string = "stream";

    stream_id:uint64;

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

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

    raw?:bytes;
}

```

A.7.9. MaxDataFrame

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

    maximum:uint64;
}
```

A.7.10. MaxStreamDataFrame

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

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

A.7.11. MaxStreamsFrame

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

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

A.7.12. DataBlockedFrame

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

    limit:uint64;
}
```

A.7.13. StreamDataBlockedFrame

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

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

A.7.14. StreamsBlockedFrame

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

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

A.7.15. NewConnectionIDFrame

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

  sequence_number:uint32;
  retire_prior_to:uint32;

  connection_id_length?:uint8;
  connection_id:bytes;

  stateless_reset_token?:Token;
}
```

A.7.16. RetireConnectionIDFrame

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

  sequence_number:uint32;
}
```

A.7.17. PathChallengeFrame

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

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

A.7.18. PathResponseFrame

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

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

## A.7.19. ConnectionCloseFrame

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

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

class ConnectionCloseFrame{
    frame_type:string = "connection_close";

    error_space?:ErrorSpace;
    error_code?:TransportError | ApplicationError | uint32;
    raw_error_code?:uint32;
    reason?:string;

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

## A.7.20. HandshakeDoneFrame

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

## A.7.21. UnknownFrame

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

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

## A.7.22. TransportError

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

#### A.7.23. CryptoError

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

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

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

## Appendix B. HTTP/3 data field definitions

### B.1. HTTP/3 Frames

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

#### B.1.1. DataFrame

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

    raw?:bytes;
}
```

#### B.1.2. HeadersFrame

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

For example:

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

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

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

#### B.1.3. CancelPushFrame

```
class CancelPushFrame{
    frame_type:string = "cancel_push";
    push_id:uint64;
}
```

#### B.1.4. SettingsFrame

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

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

#### B.1.5. PushPromiseFrame

```
class PushPromiseFrame{
    frame_type:string = "push_promise";
    push_id:uint64;

    headers:Array<HTTPHeader>;
}
```

B.1.6. GoAwayFrame

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

B.1.7. MaxPushIDFrame

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

B.1.8. DuplicatePushFrame

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

B.1.9. ReservedFrame

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

B.1.10. UnknownFrame

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

B.2. ApplicationError

```
enum ApplicationError{
    http_no_error,
    http_general_protocol_error,
    http_internal_error,
    http_stream_creation_error,
    http_closed_critical_stream,
    http_frame_unexpected,
    http_frame_error,
    http_excessive_load,
    http_id_error,
    http_settings_error,
    http_missing_settings,
    http_request_rejected,
    http_request_cancelled,
    http_request_incomplete,
    http_early_response,
    http_connect_error,
    http_version_fallback
}
```

## Appendix C. QPACK DATA type definitions

### C.1. QPACK Instructions

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

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

#### C.1.1. SetDynamicTableCapacityInstruction

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

    capacity:uint32;
}
```

#### C.1.2. InsertWithNameReferenceInstruction



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

    table_type:"static"|"dynamic";

    name_index:uint32;

    huffman_encoded_value:boolean;

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

C.1.3. InsertWithoutNameReferenceInstruction

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

    huffman_encoded_name:boolean;

    name_length?:uint32;
    name?:string;

    huffman_encoded_value:boolean;

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

C.1.4. DuplicateInstruction

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

    index:uint32;
}
```

C.1.5. HeaderAcknowledgementInstruction

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

    stream_id:uint64;
}
```

C.1.6. StreamCancellationInstruction

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

    stream_id:uint64;
}
```

#### C.1.7. InsertCountIncrementInstruction

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

    increment:uint32;
}
```

#### C.2. QPACK Header compression

```
type QPackHeaderBlockRepresentation = IndexedHeaderField | LiteralHeaderFieldWith
Name | LiteralHeaderFieldWithoutName;
```

##### C.2.1. IndexedHeaderField

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

```
class IndexedHeaderField {
    header_field_type:string = "indexed_header";

    table_type:"static"|"dynamic"; // MUST be "dynamic" if is_post_base is true
    index:uint32;

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

##### C.2.2. LiteralHeaderFieldWithName

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

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

    preserve_literal:boolean; // the 3rd "N" bit
    table_type:"static"|"dynamic"; // MUST be "dynamic" if is_post_base is true
    name_index:uint32;

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

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

#### C.2.3. LiteralHeaderFieldWithoutName

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

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

    huffman_encoded_name:boolean;
    name_length?:uint32;
    name?:string;

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

#### C.2.4. QPackHeaderBlockPrefix

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

### Appendix D. Change Log

#### D.1. Since draft-01:

##### Major changes:

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

- \* Moved packet\_type fields to PacketHeader. Moved packet\_size field out of PacketHeader to RawInfo:length (#40)
- \* Made events that need to log packet\_type and packet\_number use a header field instead of logging these fields individually
- \* Added support for logging retry, stateless reset and initial tokens (#94,#86,#117)
- \* Moved separate general event categories into a single category "generic" (#47)
- \* Added "transport:connection\_closed" event (#43,#85,#78,#49)
- \* Added version\_information and alpn\_information events (#85,#75,#28)
- \* Added parameters\_restored events to help clarify 0-RTT behaviour (#88)

Smaller changes:

- \* Merged loss\_timer events into one loss\_timer\_updated event
- \* Field data types are now strongly defined (#10,#39,#36,#115)
- \* Renamed qpack instruction\_received and instruction\_sent to instruction\_created and instruction\_parsed (#114)
- \* Updated qpack:dynamic\_table\_updated.update\_type. It now has the value "inserted" instead of "added" (#113)
- \* Updated qpack:dynamic\_table\_updated. It now has an "owner" field to differentiate encoder vs decoder state (#112)
- \* Removed push\_allowed from http:parameters\_set (#110)
- \* Removed explicit trigger field indications from events, since this was moved to be a generic property of the "data" field (#80)
- \* Updated transport:connection\_id\_updated to be more in line with other similar events. Also dropped importance from Core to Base (#45)
- \* Added length property to PaddingFrame (#34)
- \* Added packet\_number field to transport:frames\_processed (#74)

- \* Added a way to generically log packet header flags (first 8 bits) to PacketHeader
- \* Added additional guidance on which events to log in which situations (#53)
- \* Added "simulation:scenario" event to help indicate simulation details
- \* Added "packets\_acked" event (#107)
- \* Added "datagram\_ids" to the datagram\_X and packet\_X events to allow tracking of coalesced QUIC packets (#91)
- \* Extended connection\_state\_updated with more fine-grained states (#49)

D.2. Since draft-00:

- \* Event and category names are now all lowercase
- \* Added many new events and their definitions
- \* "type" fields have been made more specific (especially important for PacketType fields, which are now called packet\_type instead of type)
- \* Events are given an importance indicator (issue #22)
- \* Event names are more consistent and use past tense (issue #21)
- \* Triggers have been redefined as properties of the "data" field and updated for most events (issue #23)

Appendix E. Design Variations

TBD

Appendix F. Acknowledgements

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

Author's Address

Robin Marx  
Hasselt University

Email: robin.marx@uhasselt.be

QUIC  
Internet-Draft  
Intended status: Standards Track  
Expires: November 16, 2021

R. Marx, Ed.  
KU Leuven  
L. Niccolini, Ed.  
Facebook  
M. Seemann, Ed.  
Protocol Labs  
May 15, 2021

Main logging schema for qlog  
draft-marx-qlog-main-schema-03

## Abstract

This document describes a high-level schema for a standardized logging format called qlog. This format allows easy sharing of data and the creation of reusable visualization and debugging tools. The high-level schema in this document is intended to be protocol-agnostic. Separate documents specify how the format should be used for specific protocol data. The schema is also format-agnostic, and can be represented in for example JSON, csv or protobuf.

## Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at <https://datatracker.ietf.org/drafts/current/>.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on November 16, 2021.

## Copyright Notice

Copyright (c) 2021 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<https://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents

carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

## Table of Contents

1. Introduction . . . . .	3
1.1. Notational Conventions . . . . .	4
2. Design goals . . . . .	5
3. The high level qlog schema . . . . .	6
3.1. summary . . . . .	7
3.2. traces . . . . .	8
3.3. Individual Trace containers . . . . .	9
3.3.1. configuration . . . . .	10
3.3.2. vantage_point . . . . .	12
3.4. Field name semantics . . . . .	14
3.4.1. timestamps . . . . .	15
3.4.2. category and event . . . . .	17
3.4.3. data . . . . .	18
3.4.4. protocol_type . . . . .	18
3.4.5. triggers . . . . .	19
3.4.6. group_id . . . . .	19
3.4.7. common_fields . . . . .	21
4. Guidelines for event definition documents . . . . .	23
4.1. Event design guidelines . . . . .	23
4.2. Event importance indicators . . . . .	24
4.3. Custom fields . . . . .	25
5. Generic events and data classes . . . . .	25
5.1. Raw packet and frame information . . . . .	26
5.2. Generic events . . . . .	27
5.2.1. error . . . . .	27
5.2.2. warning . . . . .	27
5.2.3. info . . . . .	27
5.2.4. debug . . . . .	28
5.2.5. verbose . . . . .	28
5.3. Simulation events . . . . .	28
5.3.1. scenario . . . . .	29
5.3.2. marker . . . . .	29
6. Serializing qlog . . . . .	29
6.1. qlog to JSON mapping . . . . .	30
6.1.1. numbers . . . . .	30
6.1.2. bytes . . . . .	31
6.1.3. Summarizing table . . . . .	32
6.1.4. Other JSON specifics . . . . .	33
6.2. qlog to NDJSON mapping . . . . .	33
6.2.1. Supporting NDJSON in tooling . . . . .	35



6.3.	Other optimized formatting options . . . . .	35
6.3.1.	Data structure optimizations . . . . .	36
6.3.2.	Compression . . . . .	37
6.3.3.	Binary formats . . . . .	37
6.3.4.	Overview and summary . . . . .	38
6.4.	Conversion between formats . . . . .	39
7.	Methods of access and generation . . . . .	40
7.1.	Set file output destination via an environment variable .	40
7.2.	Access logs via a well-known endpoint . . . . .	41
8.	Tooling requirements . . . . .	42
9.	Security and privacy considerations . . . . .	42
10.	IANA Considerations . . . . .	43
11.	References . . . . .	43
11.1.	Normative References . . . . .	43
11.2.	Informative References . . . . .	43
11.3.	URIs . . . . .	44
Appendix A.	Change Log . . . . .	45
A.1.	Since draft-marx-qlog-main-schema-draft-02: . . . . .	45
A.2.	Since draft-marx-qlog-main-schema-01: . . . . .	45
A.3.	Since draft-marx-qlog-main-schema-00: . . . . .	46
Appendix B.	Design Variations . . . . .	46
Appendix C.	Acknowledgements . . . . .	46
Authors' Addresses	. . . . .	46

## 1. Introduction

There is currently a lack of an easily usable, standardized endpoint logging format. Especially for the use case of debugging and evaluating modern Web protocols and their performance, it is often difficult to obtain structured logs that provide adequate information for tasks like problem root cause analysis.

This document aims to provide a high-level schema and harness that describes the general layout of an easily usable, shareable, aggregatable and structured logging format. This high-level schema is protocol agnostic, with logging entries for specific protocols and use cases being defined in other documents (see for example [QLOG-QUIC] for QUIC and [QLOG-H3] for HTTP/3 and QPACK-related event definitions).

The goal of this high-level schema is to provide amenities and default characteristics that each logging file should contain (or should be able to contain), such that generic and reusable toolsets can be created that can deal with logs from a variety of different protocols and use cases.

As such, this document contains concepts such as versioning, metadata inclusion, log aggregation, event grouping and log file size reduction techniques.

Feedback and discussion are welcome at <https://github.com/quiclog/internet-drafts> [1]. Readers are advised to refer to the "editor's draft" at that URL for an up-to-date version of this document.

### 1.1. Notational Conventions

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

While the qlog schema's are format-agnostic, for readability the qlog documents will use a JSON-inspired format ([RFC8259]) for examples and definitions.

As qlog can be serialized both textually but also in binary, we employ a custom datatype definition language, inspired loosely by the "TypeScript" language [2].

This document describes how to employ JSON and NDJSON as textual serializations for qlog in Section 6. Other documents will describe how to utilize other concrete serialization options, though tips and requirements for these are also listed in this document (Section 6).

The main general conventions in this document a reader should be aware of are:

- o `obj?` : this object is optional
- o `type1 | type2` : a union of these two types (object can be either `type1` OR `type2`)
- o `obj:type` : this object has this concrete type
- o `obj:array<type>` : this object is an array of this type
- o `class` : defines a new type
- o `//` : single-line comment

The main data types are:

- o `int8` : signed 8-bit integer
- o `int16` : signed 16-bit integer

- o int32 : signed 32-bit integer
- o int64 : signed 64-bit integer
- o uint8 : unsigned 8-bit integer
- o uint16 : unsigned 16-bit integer
- o uint32 : unsigned 32-bit integer
- o uint64 : unsigned 64-bit integer
- o float : 32-bit floating point value
- o double : 64-bit floating point value
- o byte : an individual raw byte (8-bit) value (use array<byte> or the shorthand "bytes" to specify a binary blob)
- o string : list of Unicode (typically UTF-8) encoded characters
- o boolean : boolean
- o enum: fixed list of values (Unless explicitly defined, the value of an enum entry is the string version of its name (e.g., initial = "initial"))
- o any : represents any object type. Mainly used here as a placeholder for more concrete types defined in related documents (e.g., specific event types)

All timestamps and time-related values (e.g., offsets) in qlog are logged as doubles in the millisecond resolution.

Other qlog documents can define their own data types (e.g., separately for each Packet type that a protocol supports).

## 2. Design goals

The main tenets for the qlog schema design are:

- o Streamable, event-based logging
- o Flexibility in the format, complexity in the tooling (e.g., few components are a MUST, tools need to deal with this)
- o Extensible and pragmatic (e.g., no complex fixed schema with extension points)

- o Aggregation and transformation friendly (e.g., the top-level element is a container for individual traces, `group_id` can be used to tag events to a particular context)
- o Metadata is stored together with event data

### 3. The high level qlog schema

A qlog file should be able to contain several individual traces and logs from multiple vantage points that are in some way related. To that end, the top-level element in the qlog schema defines only a small set of "header" fields and an array of component traces. For this document, the required "qlog\_version" field MUST have a value of "qlog-03-WIP".

Note: there have been several previously broadly deployed qlog versions based on older drafts of this document (see draft-marx-qlog-main-schema). The old values for the "qlog\_version" field were "draft-00", "draft-01" and "draft-02". When qlog was moved to the QUIC working group, we decided to increment the existing counter, rather than reverting back to -00. As such, any numbering indicating in the "qlog\_version" field is explicitly not tied to a particular version of the draft documents.

As qlog can be serialized in a variety of ways, the "qlog\_format" field is used to indicate which serialization option was chosen. Its value MUST either be one of the options defined in this document (e.g., Section 6) or the field must be omitted entirely, in which case it assumes the default value of "JSON".

In order to make it easier to parse and identify qlog files and their serialization format, the "qlog\_version" and "qlog\_format" fields and their values SHOULD be in the first 256 characters/bytes of the resulting log file.

An example of the qlog file's top-level structure is shown in Figure 1.

Definition:

```
class QlogFile {  
    qlog_version:string,  
    qlog_format?:string,  
    title?:string,  
    description?:string,  
    summary?: Summary,  
    traces: array<Trace|TraceError>  
}
```

JSON serialization:

```
{  
    "qlog_version": "draft-03-WIP",  
    "qlog_format": "JSON",  
    "title": "Name of this particular qlog file (short)",  
    "description": "Description for this group of traces (long)",  
    "summary": {  
        ...  
    },  
    "traces": [...]  
}
```

Figure 1: Top-level element

### 3.1. summary

In a real-life deployment with a large amount of generated logs, it can be useful to sort and filter logs based on some basic summarized or aggregated data (e.g., log length, packet loss rate, log location, presence of error events, ...). The summary field (if present) SHOULD be on top of the qlog file, as this allows for the file to be processed in a streaming fashion (i.e., the implementation could just read up to and including the summary field and then only load the full logs that are deemed interesting by the user).

As the summary field is highly deployment-specific, this document does not specify any default fields or their semantics. Some examples of potential entries are shown in Figure 2.

Definition (purely illustrative example):

```
class Summary {  
    "trace_count":uint32, // amount of traces in this file  
    "max_duration":uint64, // time duration of the longest trace in ms  
    "max_outgoing_loss_rate":float, // highest loss rate for outgoing packets over  
    // all traces  
    "total_event_count":uint64, // total number of events across all traces,  
    "error_count":uint64 // total number of error events in this trace  
}
```

JSON serialization:

```
{  
    "trace_count": 1,  
    "max_duration": 5006,  
    "max_outgoing_loss_rate": 0.013,  
    "total_event_count": 568,  
    "error_count": 2  
}
```

Figure 2: Summary example definition

### 3.2. traces

It is often advantageous to group several related qlog traces together in a single file. For example, we can simultaneously perform logging on the client, on the server and on a single point on their common network path. For analysis, it is useful to aggregate these three individual traces together into a single file, so it can be uniquely stored, transferred and annotated.

As such, the "traces" array contains a list of individual qlog traces. Typical qlogs will only contain a single trace in this array. These can later be combined into a single qlog file by taking the "traces" entry/entries for each qlog file individually and copying them to the "traces" array of a new, aggregated qlog file. This is typically done in a post-processing step.

The "traces" array can thus contain both normal traces (for the definition of the Trace type, see Section 3.3), but also "error" entries. These indicate that we tried to find/convert a file for inclusion in the aggregated qlog, but there was an error during the process. Rather than silently dropping the erroneous file, we can opt to explicitly include it in the qlog file as an entry in the "traces" array, as shown in Figure 3.

Definition:

```
class TraceError {  
    error_description: string, // A description of the error  
    uri?: string, // the original URI at which we attempted to find the file  
    vantage_point?: VantagePoint // see {{vantage_point}}: the vantage point we w  
ere expecting to include here  
}
```

JSON serialization:

```
{  
  "error_description": "File could not be found",  
  "uri": "/srv/traces/today/latest.qlog",  
  "vantage_point": { type: "server" }  
}
```

Figure 3: TraceError definition

Note that another way to combine events of different traces in a single qlog file is through the use of the "group\_id" field, discussed in Section 3.4.6.

### 3.3. Individual Trace containers

The exact conceptual definition of a Trace can be fluid. For example, a trace could contain all events for a single connection, for a single endpoint, for a single measurement interval, for a single protocol, etc. As such, a Trace container contains some metadata in addition to the logged events, see Figure 4.

In the normal use case however, a trace is a log of a single data flow collected at a single location or vantage point. For example, for QUIC, a single trace only contains events for a single logical QUIC connection for either the client or the server.

The semantics and context of the trace can mainly be deduced from the entries in the "common\_fields" list and "vantage\_point" field.

Definition:

```
class Trace {  
    title?: string,  
    description?: string,  
    configuration?: Configuration,  
    common_fields?: CommonFields,  
    vantage_point: VantagePoint,  
    events: array<Event>  
}
```

JSON serialization:

```
{  
  "title": "Name of this particular trace (short)",  
  "description": "Description for this trace (long)",  
  "configuration": {  
    "time_offset": 150  
  },  
  "common_fields": {  
    "ODCID": "abcde1234",  
    "time_format": "absolute"  
  },  
  "vantage_point": {  
    "name": "backend-67",  
    "type": "server"  
  },  
  "events": [...]  
}
```

Figure 4: Trace container definition

### 3.3.1. configuration

We take into account that a qlog file is usually not used in isolation, but by means of various tools. Especially when aggregating various traces together or preparing traces for a demonstration, one might wish to persist certain tool-based settings inside the qlog file itself. For this, the configuration field is used.

The configuration field can be viewed as a generic metadata field that tools can fill with their own fields, based on per-tool logic. It is best practice for tools to prefix each added field with their tool name to prevent collisions across tools. This document only defines two optional, standard, tool-independent configuration settings: "time\_offset" and "original\_uris".



Definition:

```
class Configuration {  
    time_offset:double, // in ms,  
    original_uris: array<string>,  
  
    // list of fields with any type  
}
```

JSON serialization:

```
{  
  "time_offset": 150, // starts 150ms after the first timestamp indicates  
  "original_uris": [  
    "https://example.org/trace1.qlog",  
    "https://example.org/trace2.qlog"  
  ]  
}
```

Figure 5: Configuration definition

#### 3.3.1.1. time\_offset

The `time_offset` field indicates by how many milliseconds the starting time of the current trace should be offset. This is useful when comparing logs taken from various systems, where clocks might not be perfectly synchronous. Users could use manual tools or automated logic to align traces in time and the found optimal offsets can be stored in this field for future usage. The default value is 0.

#### 3.3.1.2. original\_uris

The `original_uris` field is used when merging multiple individual qlog files or other source files (e.g., when converting .pcaps to qlog). It allows to keep better track where certain data came from. It is a simple array of strings. It is an array instead of a single string, since a single qlog trace can be made up out of an aggregation of multiple component qlog traces as well. The default value is an empty array.

#### 3.3.1.3. custom fields

Tools can add optional custom metadata to the "configuration" field to store state and make it easier to share specific data viewpoints and view configurations.

Two examples from the qvis toolset [3] are shown in Figure 6.

```

{
  "configuration" : {
    "qvis" : {
      // when loaded into the qvis toolsuite's congestion graph tool
      // zoom in on the period between 1s and 2s and select the 124th event
      defined in this trace
      "congestion_graph": {
        "startX": 1000,
        "endX": 2000,
        "focusOnEventIndex": 124
      }

      // when loaded into the qvis toolsuite's sequence diagram tool
      // automatically scroll down the timeline to the 555th event defined
      in this trace
      "sequence_diagram" : {
        "focusOnEventIndex": 555
      }
    }
  }
}

```

Figure 6: Custom configuration fields example

### 3.3.2. vantage\_point

The `vantage_point` field describes the vantage point from which the trace originates, see Figure 7. Each trace can have only a single `vantage_point` and thus all events in a trace MUST BE from the perspective of this `vantage_point`. To include events from multiple `vantage_points`, implementers can for example include multiple traces, split by `vantage_point`, in a single qlog file.

Definition:

```
class VantagePoint {
  name?: string,
  type: VantagePointType,
  flow?: VantagePointType
}

class VantagePointType {
  server, // endpoint which initiates the connection.
  client, // endpoint which accepts the connection.
  network, // observer in between client and server.
  unknown
}
```

JSON serialization examples:

```
{
  "name": "aioquic client",
  "type": "client",
}

{
  "name": "wireshark trace",
  "type": "network",
  "flow": "client"
}
```

Figure 7: VantagePoint definition

The flow field is only required if the type is "network" (for example, the trace is generated from a packet capture). It is used to disambiguate events like "packet sent" and "packet received". This is indicated explicitly because for multiple reasons (e.g., privacy) data from which the flow direction can be otherwise inferred (e.g., IP addresses) might not be present in the logs.

Meaning of the different values for the flow field: \* "client" indicates that this vantage point follows client data flow semantics (a "packet sent" event goes in the direction of the server). \* "server" indicates that this vantage point follow server data flow semantics (a "packet sent" event goes in the direction of the client). \* "unknown" indicates that the flow's direction is unknown.

Depending on the context, tools confronted with "unknown" values in the vantage\_point can either try to heuristically infer the semantics from protocol-level domain knowledge (e.g., in QUIC, the client

always sends the first packet) or give the user the option to switch between client and server perspectives manually.

### 3.4. Field name semantics

Inside of the "events" field of a qlog trace is a list of events logged by the endpoint. Each event is specified as a generic object with a number of member fields and their associated data. Depending on the protocol and use case, the exact member field names and their formats can differ across implementations. This section lists the main, pre-defined and reserved field names with specific semantics and expected corresponding value formats.

Each qlog event at minimum requires the "time" (Section 3.4.1), "name" (Section 3.4.2) and "data" (Section 3.4.3) fields. Other typical fields are "time\_format" (Section 3.4.1), "protocol\_type" (Section 3.4.4), "trigger" (Section 3.4.5), and "group\_id" (Section 3.4.6). As especially these later fields typically have identical values across individual event instances, they are normally logged separately in the "common\_fields" (Section 3.4.7).

The specific values for each of these fields and their semantics are defined in separate documents, specific per protocol or use case. For example: event definitions for QUIC, HTTP/3 and QPACK can be found in [QLOG-QUIC] and [QLOG-H3].

Other fields are explicitly allowed by the qlog approach, and tools SHOULD allow for the presence of unknown event fields, but their semantics depend on the context of the log usage (e.g., for QUIC, the ODCID field is used), see [QLOG-QUIC].

An example of a qlog event with its component fields is shown in Figure 8.

Definition:

```
class Event {
  time: double,
  name: string,
  data: any,

  protocol_type?: Array<string>,
  group_id?: string|uint32,

  time_format?: "absolute"|"delta"|"relative",

  // list of fields with any type
}
```

JSON serialization:

```
{
  time: 1553986553572,

  name: "transport:packet_sent",
  data: { ... }

  protocol_type: ["QUIC","HTTP3"],
  group_id: "127ecc830d98f9d54a42c4f0842aa87e181a",

  time_format: "absolute",

  ODCID: "127ecc830d98f9d54a42c4f0842aa87e181a", // QUIC specific
}
```

Figure 8: Event fields definition

#### 3.4.1. timestamps

The "time" field indicates the timestamp at which the event occurred. Its value is typically the Unix timestamp since the 1970 epoch (number of milliseconds since midnight UTC, January 1, 1970, ignoring leap seconds). However, qlog supports two more succinct timestamps formats to allow reducing file size. The employed format is indicated in the "time\_format" field, which allows one of three values: "absolute", "delta" or "relative":

- o Absolute: Include the full absolute timestamp with each event. This approach uses the largest amount of characters. This is also the default value of the "time\_format" field.

- o **Delta:** Delta-encode each time value on the previously logged value. The first event in a trace typically logs the full absolute timestamp. This approach uses the least amount of characters.
- o **Relative:** Specify a full "reference\_time" timestamp (typically this is done up-front in "common\_fields", see Section 3.4.7) and include only relatively-encoded values based on this reference\_time with each event. The "reference\_time" value is typically the first absolute timestamp. This approach uses a medium amount of characters.

The first option is good for stateless loggers, the second and third for stateful loggers. The third option is generally preferred, since it produces smaller files while being easier to reason about. An example for each option can be seen in Figure 9.

The absolute approach will use:  
1500, 1505, 1522, 1588

The delta approach will use:  
1500, 5, 17, 66

The relative approach will:  
- set the reference\_time to 1500 in "common\_fields"  
- use: 0, 5, 22, 88

Figure 9: Three different approaches for logging timestamps

One of these options is typically chosen for the entire trace (put differently: each event has the same value for the "time\_format" field). Each event **MUST** include a timestamp in the "time" field.

Events in each individual trace **SHOULD** be logged in strictly ascending timestamp order (though not necessarily absolute value, for the "delta" format). Tools **CAN** sort all events on the timestamp before processing them, though are not required to (as this could impose a significant processing overhead). This can be a problem especially for multi-threaded and/or streaming loggers, who could consider using a separate postprocessor to order qlog events in time if a tool do not provide this feature.

Timestamps do not have to use the UNIX epoch timestamp as their reference. For example for privacy considerations, any initial reference timestamps (for example "endpoint uptime in ms" or "time since connection start in ms") can be chosen. Tools **SHOULD NOT** assume the ability to derive the absolute Unix timestamp from qlog

traces, nor allow on them to relatively order events across two or more separate traces (in this case, clock drift should also be taken into account).

### 3.4.2. category and event

Events differ mainly in the type of metadata associated with them. To help identify a given event and how to interpret its metadata in the "data" field (see Section 3.4.3), each event has an associated "name" field. This can be considered as a concatenation of two other fields, namely event "category" and event "type".

Category allows a higher-level grouping of events per specific event type. For example for QUIC and HTTP/3, the different categories could be "transport", "http", "qpack", and "recovery". Within these categories, the event Type provides additional granularity. For example for QUIC and HTTP/3, within the "transport" Category, there would be "packet\_sent" and "packet\_received" events.

Logging category and type separately conceptually allows for fast and high-level filtering based on category and the re-use of event types across categories. However, it also considerably inflates the log size and this flexibility is not used extensively in practice at the time of writing.

As such, the default approach in qlog is to concatenate both field values using the ":" character in the "name" field, as can be seen in Figure 10. As such, qlog category and type names MUST NOT include this character.

JSON serialization using separate fields:

```
{
  category: "transport",
  type: "packet_sent"
}
```

JSON serialization using ":" concatenated field:

```
{
  name: "transport:packet_sent"
}
```

Figure 10: Ways of logging category, type and name of an event.

Certain serializations CAN emit category and type as separate fields, and qlog tools SHOULD be able to deal with both the concatenated "name" field, and the separate "category" and "type" fields. Text-based serializations however are encouraged to employ the concatenated "name" field for efficiency.

### 3.4.3. data

The data field is a generic object. It contains the per-event metadata and its form and semantics are defined per specific sort of event. For example, data field value definitions for QUIC and HTTP/3 can be found in [QLOG-QUIC] and [QLOG-H3].

One purely illustrative example for a QUIC "packet\_sent" event is shown in Figure 11.

Definition:

```
class TransportPacketSentEvent {  
    packet_size?:uint32,  
    header:PacketHeader,  
    frames?:Array<QuicFrame>  
}
```

JSON serialization:

```
{  
  packet_size: 1280,  
  header: {  
    packet_type: "1RTT",  
    packet_number: 123  
  },  
  frames: [  
    {  
      frame_type: "stream",  
      length: 1000,  
      offset: 456  
    },  
    {  
      frame_type: "padding"  
    }  
  ]  
}
```

Figure 11: Example of the 'data' field for a QUIC packet\_sent event

### 3.4.4. protocol\_type

The "protocol\_type" array field indicates to which protocols (or protocol "stacks") this event belongs. This allows a single qlog file to aggregate traces of different protocols (e.g., a web server offering both TCP+HTTP/2 and QUIC+HTTP/3 connections).



For example, QUIC and HTTP/3 events have the "QUIC" and "HTTP3" protocol\_type entry values, see [QLOG-QUIC] and [QLOG-H3].

Typically however, all events in a single trace are of the same few protocols, and this array field is logged once in "common\_fields", see Section 3.4.7.

#### 3.4.5. triggers

Sometimes, additional information is needed in the case where a single event can be caused by a variety of other events. In the normal case, the context of the surrounding log messages gives a hint as to which of these other events was the cause. However, in highly-parallel and optimized implementations, corresponding log messages might be separated in time. Another option is to explicitly indicate these "triggers" in a high-level way per-event to get more fine-grained information without much additional overhead.

In qlog, the optional "trigger" field contains a string value describing the reason (if any) for this event instance occurring. While this "trigger" field could be a property of the qlog Event itself, it is instead a property of the "data" field instead. This choice was made because many event types do not include a trigger value, and having the field at the Event-level would cause overhead in some serializations. Additional information on the trigger can be added in the form of additional member fields of the "data" field value, yet this is highly implementation-specific, as are the trigger field's string values.

One purely illustrative example of some potential triggers for QUIC's "packet\_dropped" event is shown in Figure 12.

Definition:

```
class QuicPacketDroppedEvent {
    packet_type?:PacketType,
    raw_length?:uint32,

    trigger?: "key_unavailable" | "unknown_connection_id" | "decrypt_error" | "un
supported_version"
}
```

Figure 12: Trigger example

#### 3.4.6. group\_id

As discussed in Section 3.3, a single qlog file can contain several traces taken from different vantage points. However, a single trace from one endpoint can also contain events from a variety of sources.

For example, a server implementation might choose to log events for all incoming connections in a single large (streamed) qlog file. As such, we need a method for splitting up events belonging to separate logical entities.

The simplest way to perform this splitting is by associating a "group identifier" to each event that indicates to which conceptual "group" each event belongs. A post-processing step can then extract events per group. However, this group identifier can be highly protocol and context-specific. In the example above, we might use QUIC's "Original Destination Connection ID" to uniquely identify a connection. As such, they might add a "ODCID" field to each event. However, a middlebox logging IP or TCP traffic might rather use four-tuples to identify connections, and add a "four\_tuple" field.

As such, to provide consistency and ease of tooling in cross-protocol and cross-context setups, qlog instead defines the common "group\_id" field, which contains a string value. Implementations are free to use their preferred string serialization for this field, so long as it contains a unique value per logical group. Some examples can be seen in Figure 13.

JSON serialization for events grouped by four tuples and QUIC connection IDs:

```
events: [
  {
    time: 1553986553579,
    protocol_type: ["TCP", "TLS", "HTTP2"],
    group_id: "ip1=2001:67c:1232:144:9498:6df6:f450:110b,ip2=2001:67c:2b0:1c1
::198,port1=59105,port2=80",
    name: "transport:packet_received",
    data: { ... },
  },
  {
    time: 1553986553581,
    protocol_type: ["QUIC", "HTTP3"],
    group_id: "127ecc830d98f9d54a42c4f0842aa87e181a",
    name: "transport:packet_sent",
    data: { ... },
  }
]
```

Figure 13: Example of group\_id usage

Note that in some contexts (for example a Multipath transport protocol) it might make sense to add additional contextual per-event fields (for example "path\_id"), rather than use the group\_id field for that purpose.

Note also that, typically, a single trace only contains events belonging to a single logical group (for example, an individual QUIC connection). As such, instead of logging the "group\_id" field with an identical value for each event instance, this field is typically logged once in "common\_fields", see Section 3.4.7.

#### 3.4.7. common\_fields

As discussed in the previous sections, information for a typical qlog event varies in three main fields: "time", "name" and associated data. Additionally, there are also several more advanced fields that allow mixing events from different protocols and contexts inside of the same trace (for example "protocol\_type" and "group\_id"). In most "normal" use cases however, the values of these advanced fields are consistent for each event instance (for example, a single trace contains events for a single QUIC connection).

To reduce file size and making logging easier, qlog uses the "common\_fields" list to indicate those fields and their values that are shared by all events in this component trace. This prevents these fields from being logged for each individual event. An example of this is shown in Figure 14.

JSON serialization with repeated field values per-event instance:

```
{
  events: [{
    group_id: "127ecc830d98f9d54a42c4f0842aa87e181a",
    protocol_type: ["QUIC", "HTTP3"],
    time_format: "relative",
    reference_time: "1553986553572",

    time: 2,
    name: "transport:packet_received",
    data: { ... }
  }, {
    group_id: "127ecc830d98f9d54a42c4f0842aa87e181a",
    protocol_type: ["QUIC", "HTTP3"],
    time_format: "relative",
    reference_time: "1553986553572",

    time: 7,
    name: "http:frame_parsed",
    data: { ... }
  }
]
```

JSON serialization with repeated field values extracted to common\_fields:

```
{
  common_fields: {
    group_id: "127ecc830d98f9d54a42c4f0842aa87e181a",
    protocol_type: ["QUIC", "HTTP3"],
    time_format: "relative",
    reference_time: "1553986553572"
  },
  events: [
    {
      time: 2,
      name: "transport:packet_received",
      data: { ... }
    }, {
      7,
      name: "http:frame_parsed",
      data: { ... }
    }
  ]
}
```

Figure 14: Example of common\_fields usage

The "common\_fields" field is a generic dictionary of key-value pairs, where the key is always a string and the value can be of any type, but is typically also a string or number. As such, unknown entries in this dictionary MUST be disregarded by the user and tools (i.e., the presence of an unknown field is explicitly NOT an error).

The list of default qlog fields that are typically logged in common\_fields (as opposed to as individual fields per event instance) are:

- o time\_format
- o reference\_time
- o protocol\_type
- o group\_id

Tools MUST be able to deal with these fields being defined either on each event individually or combined in common\_fields. Note that if at least one event in a trace has a different value for a given field, this field MUST NOT be added to common\_fields but instead defined on each event individually. Good example of such fields are "time" and "data", who are divergent by nature.

#### 4. Guidelines for event definition documents

This document only defines the main schema for the qlog format. This is intended to be used together with specific, per-protocol event definitions that specify the name (category + type) and data needed for each individual event. This is with the intent to allow the qlog main schema to be easily re-used for several protocols. Examples include the QUIC event definitions [QLOG-QUIC] and HTTP/3 and QPACK event definitions [QLOG-H3].

This section defines some basic annotations and concepts the creators of event definition documents SHOULD follow to ensure a measure of consistency, making it easier for qlog implementers to extrapolate from one protocol to another.

##### 4.1. Event design guidelines

TODO: pending QUIC working group discussion. This text reflects the initial (qlog draft 01 and 02) setup.

There are several ways of defining qlog events. In practice, we have seen two main types used so far: a) those that map directly to concepts seen in the protocols (e.g., "packet\_sent") and b) those

that act as aggregating events that combine data from several possible protocol behaviours or code paths into one (e.g., "parameters\_set"). The latter are typically used as a means to reduce the amount of unique event definitions, as reflecting each possible protocol event as a separate qlog entity would cause an explosion of event types.

Additionally, logging duplicate data is typically prevented as much as possible. For example, packet header values that remain consistent across many packets are split into separate events (for example "spin\_bit\_updated" or "connection\_id\_updated" for QUIC).

Finally, we have typically refrained from adding additional state change events if those state changes can be directly inferred from data on the wire (for example flow control limit changes) if the implementation is bug-free and spec-compliant. Exceptions have been made for common events that benefit from being easily identifiable or individually logged (for example "packets\_acked").

#### 4.2. Event importance indicators

Depending on how events are designed, it may be that several events allow the logging of similar or overlapping data. For example the separate QUIC "connection\_started" event overlaps with the more generic "connection\_state\_updated". In these cases, it is not always clear which event should be logged or used, and which event should take precedence if e.g., both are present and provide conflicting information.

To aid in this decision making, we recommend that each event SHOULD have an "importance indicator" with one of three values, in decreasing order of importance and expected usage:

- o Core
- o Base
- o Extra

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

The "Base" events add additional debugging options and CAN be present in qlog files. Most of these can be implicitly inferred from data in

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

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

Note that in some cases, implementers might not want to log for example data content details in the "Core" events due to performance or privacy considerations. In this case, they SHOULD use (a subset of) relevant "Base" events instead to ensure usability of the qlog output. As an example, implementations that do not log QUIC "packet\_received" events and thus also not which (if any) ACK frames the packet contains, SHOULD log "packets\_acked" events instead.

Finally, for event types whose data (partially) overlap with other event types' definitions, where necessary the event definition document should include explicit guidance on which to use in specific situations.

#### 4.3. Custom fields

Event definition documents are free to define new category and event types, top-level fields (e.g., a per-event field indicating its privacy properties or path\_id in multipath protocols), as well as values for the "trigger" property within the "data" field, or other member fields of the "data" field, as they see fit.

They however SHOULD NOT expect non-specialized tools to recognize or visualize this custom data. However, tools SHOULD make an effort to visualize even unknown data if possible in the specific tool's context. If they do not, they MUST ignore these unknown fields.

#### 5. Generic events and data classes

There are some event types and data classes that are common across protocols, applications and use cases that benefit from being defined in a single location. This section specifies such common definitions.

### 5.1. Raw packet and frame information

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

```
class RawInfo {  
    length?:uint64; // the full byte length of the entity (e.g., packet or frame)  
    including headers and trailers  
    payload_length?:uint64; // the byte length of the entity's payload, without h  
    eaders or trailers  
  
    data?:bytes; // the contents of the full entity, including headers and traile  
rs  
}
```

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

Note: We do not specify explicit header\_length or trailer\_length fields. In most protocols, header\_length can be calculated by subtracing the payload\_length from the length (e.g., if trailer\_length is always 0). In protocols with trailers (e.g., QUIC's AEAD tag), event definitions documents SHOULD define other ways of logging the trailer\_length to make the header\_length calculation possible.

The exact definitions entities, headers, trailers and payloads depend on the protocol used. If this is non-trivial, event definitions documents SHOULD include a clear explanation of how entities are mapped into the RawInfo structure.

Note: Relatedly, many modern protocols use Variable-Length Integer Encoded (VLIE) values in their headers, which are of a dynamic length. Because of this, we cannot deterministically reconstruct the header encoding/length from non-RawInfo qlog data, as implementations might not necessarily employ the most efficient VLIE scheme for all values. As such, to make exact size-analysis possible, implementers should use explicit lengths in RawInfo rather than reconstructing them from other qlog data. Similarly, tool developers should only utilize RawInfo (and related information) in such tools to prevent errors.



## 5.2. Generic events

In typical logging setups, users utilize a discrete number of well-defined logging categories, levels or severities to log freeform (string) data. This generic events category replicates this approach to allow implementations to fully replace their existing text-based logging by qlog. This is done by providing events to log generic strings for the typical well-known logging levels (error, warning, info, debug, verbose).

For the events defined below, the "category" is "generic" and their "type" is the name of the heading in lowercase (e.g., the "name" of the error event is "generic:error").

### 5.2.1. error

Importance: Core

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

Data:

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

### 5.2.2. warning

Importance: Base

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

Data:

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

### 5.2.3. info

Importance: Extra

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

Data:

```
{  
    message:string  
}
```

#### 5.2.4. debug

Importance: Extra

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

Data:

```
{  
    message:string  
}
```

#### 5.2.5. verbose

Importance: Extra

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

Data:

```
{  
    message:string  
}
```

### 5.3. Simulation events

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

For the events defined below, the "category" is "simulation" and their "type" is the name of the heading in lowercase (e.g., the "name" of the scenario event is "simulation:scenario").

#### 5.3.1. scenario

Importance: Extra

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

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

#### 5.3.2. marker

Importance: Extra

Used to indicate when specific emulation conditions are triggered at set times (e.g., at 3 seconds in 2% packet loss is introduced, at 10s a NAT rebind is triggered).

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

### 6. Serializing qlog

This document and other related qlog schema definitions are intentionally serialization-format agnostic. This means that implementers themselves can choose how to represent and serialize qlog data practically on disk or on the wire. Some examples of possible formats are JSON, CBOR, CSV, protocol buffers, flatbuffers, etc.

All these formats make certain tradeoffs between flexibility and efficiency, with textual formats like JSON typically being more flexible but also less efficient than binary formats like protocol buffers. The format choice will depend on the practical use case of the qlog user. For example, for use in day to day debugging, a plaintext readable (yet relatively large) format like JSON is probably preferred. However, for use in production, a more optimized

yet restricted format can be better. In this latter case, it will be more difficult to achieve interoperability between qlog implementations of various protocol stacks, as some custom or tweaked events from one might not be compatible with the format of the other. This will also reflect in tooling: not all tools will support all formats.

This being said, the authors prefer JSON as the basis for storing qlog, as it retains full flexibility and maximum interoperability. Storage overhead can be managed well in practice by employing compression. For this reason, this document details both how to practically transform qlog schema definitions to JSON and to the streamable NDJSON. We discuss concrete options to bring down JSON size and processing overheads in Section 6.3.

As depending on the employed format different deserializers/parsers should be used, the "qlog\_format" field is used to indicate the chosen serialization approach. This field is always a string, but can be made hierarchical by the use of the "." separator between entries. For example, a value of "JSON.optimizationA" can indicate that a default JSON format is being used, but that a certain optimization of type A was applied to the file as well (see also Section 6.3).

#### 6.1. qlog to JSON mapping

When mapping qlog to normal JSON, the "qlog\_format" field MUST have the value "JSON". This is also the default qlog serialization and default value of this field.

To facilitate this mapping, the qlog documents employ a format that is close to pure JSON for its examples and data definitions. Still, as JSON is not a typed format, there are some practical peculiarities to observe.

##### 6.1.1. numbers

While JSON has built-in support for integers up to 64 bits in size, not all JSON parsers do. For example, none of the major Web browsers support full 64-bit integers at this time, as all numerical values (both floating-point numbers and integers) are internally represented as floating point IEEE 754 [4] values. In practice, this limits their integers to a maximum value of  $2^{53}-1$ . Integers larger than that are either truncated or produce a JSON parsing error. While this is expected to improve in the future (as "BigInt" support [5] has been introduced in most Browsers, though not yet integrated into JSON parsers), we still need to deal with it here.

When transforming an `int64`, `uint64` or `double` from qlog to JSON, the implementer can thus choose to either log them as JSON numbers (taking the risk of truncation or un-parseability) or to log them as strings instead. Logging as strings should however only be practically needed if the value is likely to exceed  $2^{53}-1$ . In practice, even though protocols such as QUIC allow 64-bit values for for example stream identifiers, these high numbers are unlikely to be reached for the overwhelming majority of cases. As such, it is probably a valid trade-off to take the risk and log 64-bit values as JSON numbers instead of strings.

Tools processing JSON-based qlog SHOULD however be able to deal with 64-bit fields being serialized as either strings or numbers.

#### 6.1.2. bytes

Unlike most binary formats, JSON does not allow the logging of raw binary blobs directly. As such, when serializing a byte or `array<byte>`, a scheme needs to be chosen.

To represent qlog bytes in JSON, they MUST be serialized to their lowercase hexadecimal equivalents (with 0 prefix for values lower than 10). All values are directly appended to each other, without delimiters. The full value is not prefixed with 0x (as is sometimes common). An example is given in Figure 15.

For the five raw unsigned byte input values of: 5 20 40 171 255, the JSON serialization is:

```
{
  raw: "051428abff"
}
```

Figure 15: Example for serializing bytes

As such, the resulting string will always have an even amount of characters and the original byte-size can be retrieved by dividing the string length by 2.

##### 6.1.2.1. Truncated values

In some cases, it can be interesting not to log a full raw blob but instead a truncated value (for example, only the first 100 bytes of an HTTP response body to be able to discern which file it actually contained). In these cases, the original byte-size length cannot be obtained from the serialized value directly. As such, all qlog schema definitions SHOULD include a separate, length-indicating field for all fields of type `array<byte>` they specify. This allows always retrieving the original length, but also allows the omission of any

raw value bytes of the field completely (e.g., out of privacy or security considerations).

To reduce overhead however and in the case the full raw value is logged, the extra length-indicating field can be left out. As such, tools MUST be able to deal with this situation and derive the length of the field from the raw value if no separate length-indicating field is present. All possible permutations are shown by example in Figure 16.

```
// both the full raw value and its length are present (length is redundant)
{
  "raw_length": 5,
  "raw": "051428abff"
}

// only the raw value is present, indicating it represents the fields full value
// the byte length is obtained by calculating raw.length / 2
{
  "raw": "051428abff"
}

// only the length field is present, meaning the value was omitted
{
  "raw_length": 5,
}

// both fields are present and the lengths do not match: the value was truncated
// to the first three bytes.
{
  "raw_length": 5,
  "raw": "051428"
}
```

Figure 16: Example for serializing truncated bytes

#### 6.1.3. Summarizing table

By definition, JSON strings are serialized surrounded by quotes. Numbers without.

qlog type	JSON type
int8	number
int16	number
int32	number
uint8	number
uint16	number
uint32	number
float	number
int64	number or string
uint64	number or string
double	number or string
bytes	string (lowercase hex value)
string	string
boolean	string ("true" or "false")
enum	string (full value/name, not index)
any	object ( {...} )
array	array ( [...] )

#### 6.1.4. Other JSON specifics

JSON files by definition ([RFC8259]) MUST utilize the UTF-8 encoding, both for the file itself and the string values.

Most JSON parsers strictly follow the JSON specification. This includes the rule that trailing comma's are not allowed. As it is frequently annoying to remove these trailing comma's when logging events in a streaming fashion, tool implementers SHOULD allow the last event entry of a qlog trace to be an empty object. This allows loggers to simply close the qlog file by appending "{}]]]]" after their last added event.

Finally, while not specifically required by the JSON specification, all qlog field names in a JSON serialization MUST be lowercase.

#### 6.2. qlog to NDJSON mapping

One of the downsides of using pure JSON is that it is inherently a non-streamable format. Put differently, it is not possible to simply append new qlog events to a log file without "closing" this file at the end by appending "]]]]". Without these closing tags, most JSON parsers will be unable to parse the file entirely. As most platforms do not provide a standard streaming JSON parser (which would be able to deal with this problem), this document also provides a qlog mapping to a streamable JSON format called Newline-Delimited JSON (NDJSON) [6].

When mapping qlog to NDJSON, the "qlog\_format" field MUST have the value "NDJSON".

NDJSON is very similar to JSON, except that it interprets each line in a file as a fully separate JSON object. Put differently, unlike default JSON, it does not require a file to be wrapped as a full object with "{ ... }" or "[ ... ]". Using this setup, qlog events can simply be appended as individually serialized lines at the back of a streamed logging file.

For this to work, some qlog definitions have to be adjusted however. Mainly, events are no longer part of the "events" array in the Trace object, but are instead logged separately from the qlog "file header" (QlogFile class in Section 3). Additionally, qlog's NDJSON mapping does not allow logging multiple individual traces in a single qlog file. As such, the QlogFile:traces field is replaced by the singular "trace" field, which simply contains the Trace data directly. An example can be seen in Figure 17. Note that the "group\_id" field can still be used on a per-event basis to include events from conceptually different sources in a single NDJSON qlog file.

Note as well from Figure 17 that the file's header (QlogFileNDJSON) also needs to be fully serialized on a single line to be NDJSON compatible.

Definition:

```
class QlogFileNDJSON {
    qlog_format: "NDJSON",

    qlog_version:string,
    title?:string,
    description?:string,
    summary?: Summary,
    trace: Trace
}
// list of qlog events, separated by newlines
```

NDJSON serialization:

```
{"qlog_format":"NDJSON","qlog_version":"draft-03-WIP","title":"Name of this parti
cular NDJSON qlog file (short)","description":"Description for this NDJSON qlog f
ile (long)","trace":{"common_fields":{"protocol_type": ["QUIC","HTTP3"],"group_id
":"127ecc830d98f9d54a42c4f0842aa87e181a","time_format":"relative","reference_time
":"1553986553572"},"vantage_point":{"name":"backend-67","type":"server"}}}
{"time": 2, "name": "transport:packet_received", "data": { ... } }
{"time": 7, "name": "http:frame_parsed", "data": { ... } }
```

Figure 17: Top-level element

Finally, while not specifically required by the NDJSON specification, all qlog field names in a NDJSON serialization MUST be lowercase.



#### 6.2.1. Supporting NDJSON in tooling

Note that NDJSON is not supported in most default programming environments (unlike normal JSON). However, several custom NDJSON parsing libraries exist [7] that can be used and the format is easy enough to parse with existing implementations (i.e., by splitting the file into its component lines and feeding them to a normal JSON parser individually, as each line by itself is a valid JSON object).

#### 6.3. Other optimized formatting options

Both the JSON and NDJSON formatting options described above are serviceable in general small to medium scale (debugging) setups. However, these approaches tend to be relatively verbose, leading to larger file sizes. Additionally, generalized (ND)JSON (de)serialization performance is typically (slightly) lower than that of more optimized and predictable formats. Both aspects make these formats more challenging (though still practical [8]) to use in large scale setups.

During the development of qlog, we compared a multitude of alternative formatting and optimization options. The results of this study are summarized on the qlog github repository [9]. The rest of this section discusses some of these approaches implementations could choose and the expected gains and tradeoffs inherent therein. Tools SHOULD support mainly the compression options listed in Section 6.3.2, as they provide the largest wins for the least cost overall.

Over time, specific qlog formats and encodings can be created that more formally define and combine some of the discussed optimizations or add new ones. We choose to define these schemes in separate documents to keep the main qlog definition clean and generalizable, as not all contexts require the same performance or flexibility as others and qlog is intended to be a broadly usable and extensible format (for example more flexibility is needed in earlier stages of protocol development, while more performance is typically needed in later stages). This is also the main reason why the general qlog format is the less optimized JSON instead of a more performant option.

To be able to easily distinguish between these options in qlog compatible tooling (without the need to have the user provide out-of-band information or to (heuristically) parse and process files in a multitude of ways, see also Section 8), we recommend using explicit file extensions to indicate specific formats. As there are no standards in place for this type of extension to format mapping, we employ a commonly used scheme here. Our approach is to list the

applied optimizations in the extension in ascending order of application (e.g., if a qlog file is first optimized with technique A and then compressed with technique B, the resulting file would have the extension ".qlog.A.B"). This allows tooling to start at the back of the extension to "undo" applied optimizations to finally arrive at the expected qlog representation.

#### 6.3.1. Data structure optimizations

The first general category of optimizations is to alter the representation of data within an (ND)JSON qlog file to reduce file size.

The first option is to employ a scheme similar to the CSV (comma separated value [rfc4180]) format, which utilizes the concept of column "headers" to prevent repeating field names for each datapoint instance. Concretely for JSON qlog, several field names are repeated with each event (i.e., time, name, data). These names could be extracted into a separate list, after which qlog events could be serialized as an array of values, as opposed to a full object. This approach was a key part of the original qlog format (prior to draft 02) using the "event\_fields" field. However, tests showed that this optimization only provided a mean file size reduction of 5% (100MB to 95MB) while significantly increasing the implementation complexity, and this approach was abandoned in favor of the default JSON setup. Implementations using this format should not employ a separate file extension (as it still uses JSON), but rather employ a new value of "JSON.namedheaders" (or "NDJSON.namedheaders") for the "qlog\_format" field (see Section 3).

The second option is to replace field values and/or names with indices into a (dynamic) lookup table. This is a common compression technique and can provide significant file size reductions (up to 50% in our tests, 100MB to 50MB). However, this approach is even more difficult to implement efficiently and requires either including the (dynamic) table in the resulting file (an approach taken by for example Chromium's NetLog format [10]) or defining a (static) table up-front and sharing this between implementations. Implementations using this approach should not employ a separate file extension (as it still uses JSON), but rather employ a new value of "JSON.dictionary" (or "NDJSON.dictionary") for the "qlog\_format" field (see Section 3).

As both options either proved difficult to implement, reduced qlog file readability, and provided too little improvement compared to other more straightforward options (for example Section 6.3.2), these schemes are not inherently part of qlog.

### 6.3.2. Compression

The second general category of optimizations is to utilize a (generic) compression scheme for textual data. As qlog in the (ND)JSON format typically contains a large amount of repetition, off-the-shelf (text) compression techniques typically succeed very well in bringing down file sizes (regularly with up to two orders of magnitude in our tests, even for "fast" compression levels). As such, utilizing compression is recommended before attempting other optimization options, even though this might (somewhat) increase processing costs due to the additional compression step.

The first option is to use GZIP compression ([RFC1952]). This generic compression scheme provides multiple compression levels (providing a trade-off between compression speed and size reduction). Utilized at level 6 (a medium setting thought to be applicable for streaming compression of a qlog stream in commodity devices), gzip compresses qlog JSON files to 7% of their initial size on average (100MB to 7MB). For this option, the file extension .qlog.gz SHOULD BE used. The "qlog\_format" field should still reflect the original JSON formatting of the qlog data (e.g., "JSON" or "NDJSON").

The second option is to use Brotli compression ([RFC7932]). While similar to gzip, this more recent compression scheme provides a better efficiency. It also allows multiple compression levels. Utilized at level 4 (a medium setting thought to be applicable for streaming compression of a qlog stream in commodity devices), brotli compresses qlog JSON files to 7% of their initial size on average (100MB to 7MB). For this option, the file extension .qlog.br SHOULD BE used. The "qlog\_format" field should still reflect the original JSON formatting of the qlog data (e.g., "JSON" or "NDJSON").

Other compression algorithms of course exist (for example xz, zstd, and lz4). We mainly recommend gzip and brotli because of their tweakable behaviour and wide support in web-based environments, which we envision as the main tooling ecosystem (see also Section 8).

### 6.3.3. Binary formats

The third general category of optimizations is to use a more optimized (often binary) format instead of the textual JSON format. This approach inherently produces smaller files and often has better (de)serialization performance. However, the resultant files are no longer human readable and some formats require hard tradeoffs between flexibility for performance.

The first option is to use the CBOR (Concise Binary Object Representation [rfc7049]) format. For our purposes, CBOR can be

viewed as a straightforward binary variant of JSON. As such, existing JSON qlog files can be trivially converted to and from CBOR (though slightly more work is needed for NDJSON qlogs). While CBOR thus does retain the full qlog flexibility, it only provides a 25% file size reduction (100MB to 75MB) compared to textual (ND)JSON. As CBOR support in programming environments is not as widespread as that of textual JSON and the format lacks human readability, CBOR was not chosen as the default qlog format. For this option, the file extension `.qlog.cbor` SHOULD BE used. The `"qlog_format"` field should still reflect the original JSON formatting of the qlog data (e.g., `"JSON"` or `"NDJSON"`).

A second option is to use a more specialized binary format, such as Protocol Buffers [11] (protobuf). This format is battle-tested, has support for optional fields and has libraries in most programming languages. Still, it is significantly less flexible than textual JSON or CBOR, as it relies on a separate, pre-defined schema (a `.proto` file). As such, it is not possible to (easily) log new event types in protobuf files without adjusting this schema as well, which has its own practical challenges. As qlog is intended to be a flexible, general purpose format, this type of format was not chosen as its basic serialization. The lower flexibility does lead to significantly reduced file sizes. Our straightforward mapping of the qlog main schema and QUIC/HTTP3 event types to protobuf created qlog files 24% as large as the raw JSON equivalents (100MB to 24MB). For this option, the file extension `.qlog.protobuf` SHOULD BE used. The `"qlog_format"` field should reflect the different internal format, for example: `"qlog_format": "protobuf"`.

Note that binary formats can (and should) also be used in conjunction with compression (see Section 6.3.2). For example, CBOR compresses well (to about 6% of the original textual JSON size (100MB to 6MB) for both gzip and brotli) and so does protobuf (5% (gzip) to 3% (brotli)). However, these gains are similar to the ones achieved by simply compressing the textual JSON equivalents directly (7%, see Section 6.3.2). As such, since compression is still needed to achieve optimal file size reductions even with binary formats, we feel the more flexible compressed textual JSON options are a better default for the qlog format in general.

#### 6.3.4. Overview and summary

In summary, textual JSON was chosen as the main qlog format due to its high flexibility and because its inefficiencies can be largely solved by the utilization of compression techniques (which are needed to achieve optimal results with other formats as well).

Still, qlog implementers are free to define other qlog formats depending on their needs and context of use. These formats should be described in their own documents, the discussion in this document mainly acting as inspiration and high-level guidance. Implementers are encouraged to add concrete qlog formats and definitions to the designated public repository [12].

The following table provides an overview of all the discussed qlog formatting options with examples:

format	qlog_format	extension
JSON Section 6.1	JSON	.qlog
NDJSON Section 6.2	NDJSON	.qlog
named headers Section 6.3.1	(ND)JSON.namedheaders	.qlog
dictionary Section 6.3.1	(ND)JSON.dictionary	.qlog
CBOR Section 6.3.3	(ND)JSON	.qlog.cbor
protobuf Section 6.3.3	protobuf	.qlog.protobuf
gzip Section 6.3.2	no change	.gz suffix
brrotli Section 6.3.2	no change	.br suffix

#### 6.4. Conversion between formats

As discussed in the previous sections, a qlog file can be serialized in a multitude of formats, each of which can conceivably be transformed into or from one another without loss of information. For example, a number of NDJSON streamed qlogs could be combined into a JSON formatted qlog for later processing. Similarly, a captured binary qlog could be transformed to JSON for easier interpretation and sharing.

Secondly, we can also consider other structured logging approaches that contain similar (though typically not identical) data to qlog, like raw packet capture files (for example .pcap files from tcpdump) or endpoint-specific logging formats (for example the NetLog format in Google Chrome). These are sometimes the only options, if an implementation cannot or will not support direct qlog output for any reason, but does provide other internal or external (e.g., SSLKEYLOGFILE export to allow decryption of packet captures) logging options. For this second category, a (partial) transformation from/to qlog can also be defined.

As such, when defining a new qlog serialization format or wanting to utilize qlog-compatible tools with existing codebases lacking qlog

support, it is recommended to define and provide a concrete mapping from one format to default JSON-serialized qlog. Several of such mappings exist. Firstly, [pcap2qlog] (<https://github.com/quiclog/pcap2qlog>) transforms QUIC and HTTP/3 packet capture files to qlog. Secondly, netlog2qlog [13] converts chromium's internal dictionary-encoded JSON format to qlog. Finally, quictrace2qlog [14] converts the older quictrace format to JSON qlog. Tools can then easily integrate with these converters (either by incorporating them directly or for example using them as a (web-based) API) so users can provide different file types with ease. For example, the qvis [15] toolsuite supports a multitude of formats and qlog serializations.

## 7. Methods of access and generation

Different implementations will have different ways of generating and storing qlogs. However, there is still value in defining a few default ways in which to steer this generation and access of the results.

### 7.1. Set file output destination via an environment variable

To provide users control over where and how qlog files are created, we define two environment variables. The first, QLOGFILE, indicates a full path to where an individual qlog file should be stored. This path MUST include the full file extension. The second, QLOGDIR, sets a general directory path in which qlog files should be placed. This path MUST include the directory separator character at the end.

In general, QLOGDIR should be preferred over QLOGFILE if an endpoint is prone to generate multiple qlog files. This can for example be the case for a QUIC server implementation that logs each QUIC connection in a separate qlog file. An alternative that uses QLOGFILE would be a QUIC server that logs all connections in a single file and uses the "group\_id" field (Section 3.4.6) to allow post-hoc separation of events.

Implementations SHOULD provide support for QLOGDIR and MAY provide support for QLOGFILE.

When using QLOGDIR, it is up to the implementation to choose an appropriate naming scheme for the qlog files themselves. The chosen scheme will typically depend on the context or protocols used. For example, for QUIC, it is recommended to use the Original Destination Connection ID (ODCID), followed by the vantage point type of the logging endpoint. Examples of all options for QUIC are shown in Figure 18.

Command: `QLOGFILE=/srv/qlogs/client.qlog quicclientbinary`

Should result in the the `quicclientbinary` executable logging a single qlog file named `client.qlog` in the `/srv/qlogs` directory.

This is for example useful in tests when the client sets up just a single connection and then exits.

Command: `QLOGDIR=/srv/qlogs/ quicserverbinary`

Should result in the `quicserverbinary` executable generating several logs files, one for each QUIC connection.

Given two QUIC connections, with ODCID values `"abcde"` and `"12345"` respectively, this would result in two files:

`/srv/qlogs/abcde_server.qlog`

`/srv/qlogs/12345_server.qlog`

Command: `QLOGFILE=/srv/qlogs/server.qlog quicserverbinary`

Should result in the the `quicserverbinary` executable logging a single qlog file named `server.qlog` in the `/srv/qlogs` directory.

Given that the server handled two QUIC connections before it was shut down, with ODCID values `"abcde"` and `"12345"` respectively, this would result in event instances in the qlog file being tagged with the `"group_id"` field with values `"abcde"` and `"12345"`.

Figure 18: Environment variable examples for a QUIC implementation

## 7.2. Access logs via a well-known endpoint

After generation, qlog implementers MAY make available generated logs and traces on an endpoint (typically the server) via the following .well-known URI:

`.well-known/qlog/IDENTIFIER.extension`

The `IDENTIFIER` variable depends on the context and the protocol. For example for QUIC, the lowercase Original Destination Connection ID (ODCID) is recommended, as it can uniquely identify a connection. Additionally, the extension depends on the chosen format (see Section 6.3.4). For example, for a QUIC connection with ODCID `"abcde"`, the endpoint for fetching its default JSON-formatted .qlog file would be:

`.well-known/qlog/abcde.qlog`

Implementers SHOULD allow users to fetch logs for a given connection on a 2nd, separate connection. This helps prevent pollution of the logs by fetching them over the same connection that one wishes to observe through the log. Ideally, for the QUIC use case, the logs should also be approachable via an HTTP/2 or HTTP/1.1 endpoint (i.e., on TCP port 443), to for example aid debugging in the case where QUIC/UDP is blocked on the network.

qlog implementers SHOULD NOT enable this .well-known endpoint in typical production settings to prevent (malicious) users from





downloading logs from other connections. Implementers are advised to disable this endpoint by default and require specific actions from the end users to enable it (and potentially qlog itself). Implementers MUST also take into account the general privacy and security guidelines discussed in Section 9 before exposing qlogs to outside actors.

## 8. Tooling requirements

Tools ingestion qlog MUST indicate which qlog version(s), qlog format(s), compression methods and potentially other input file formats (for example .pcap) they support. Tools SHOULD at least support .qlog files in the default JSON format (Section 6.1). Additionally, they SHOULD indicate exactly which values for and properties of the name (category and type) and data fields they look for to execute their logic. Tools SHOULD perform a (high-level) check if an input qlog file adheres to the expected qlog schema. If a tool determines a qlog file does not contain enough supported information to correctly execute the tool's logic, it SHOULD generate a clear error message to this effect.

Tools MUST NOT produce breaking errors for any field names and/or values in the qlog format that they do not recognize. Tools SHOULD indicate even unknown event occurrences within their context (e.g., marking unknown events on a timeline for manual interpretation by the user).

Tool authors should be aware that, depending on the logging implementation, some events will not always be present in all traces. For example, using a circular logging buffer of a fixed size, it could be that the earliest events (e.g., connection setup events) are later overwritten by "newer" events. Alternatively, some events can be intentionally omitted out of privacy or file size considerations. Tool authors are encouraged to make their tools robust enough to still provide adequate output for incomplete logs.

## 9. Security and privacy considerations

TODO : discuss privacy and security considerations (e.g., what NOT to log, what to strip out of a log before sharing, ...)

TODO: strip out/don't log IPs, ports, specific CIDs, raw user data, exact times, HTTP HEADERS (or at least :path), SNI values

TODO: see if there is merit in encrypting the logs and having the server choose an encryption key (e.g., sent in transport parameters)

Good initial reference: Christian Huitema's blogpost [16]

## 10. IANA Considerations

TODO: primarily the .well-known URI

## 11. References

### 11.1. Normative References

[QLOG-H3] Marx, R., Ed., Niccolini, L., Ed., and M. Seemann, Ed., "HTTP/3 and QPACK event definitions for qlog", draft-marx-qlog-h3-events-00 (work in progress).

[QLOG-QUIC] Marx, R., Ed., Niccolini, L., Ed., and M. Seemann, Ed., "QUIC event definitions for qlog", draft-marx-qlog-quic-events-00 (work in progress).

### 11.2. Informative References

[RFC1952] Deutsch, P., "GZIP file format specification version 4.3", RFC 1952, DOI 10.17487/RFC1952, May 1996, <<https://www.rfc-editor.org/info/rfc1952>>.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <<https://www.rfc-editor.org/info/rfc2119>>.

[rfc4180] Shafranovich, Y., "Common Format and MIME Type for Comma-Separated Values (CSV) Files", RFC 4180, DOI 10.17487/RFC4180, October 2005, <<https://www.rfc-editor.org/info/rfc4180>>.

[rfc7049] Bormann, C. and P. Hoffman, "Concise Binary Object Representation (CBOR)", RFC 7049, DOI 10.17487/RFC7049, October 2013, <<https://www.rfc-editor.org/info/rfc7049>>.

[RFC7932] Alakuijala, J. and Z. Szabadka, "Brotli Compressed Data Format", RFC 7932, DOI 10.17487/RFC7932, July 2016, <<https://www.rfc-editor.org/info/rfc7932>>.

[RFC8259] Bray, T., Ed., "The JavaScript Object Notation (JSON) Data Interchange Format", STD 90, RFC 8259, DOI 10.17487/RFC8259, December 2017, <<https://www.rfc-editor.org/info/rfc8259>>.

### 11.3. URIs

- [1] <https://github.com/quiclog/internet-drafts>
- [2] <https://www.typescriptlang.org/>
- [3] <https://qvis.edm.uhasselt.be>
- [4] [https://en.wikipedia.org/wiki/Floating-point\\_arithmetic](https://en.wikipedia.org/wiki/Floating-point_arithmetic)
- [5] [https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\\_Objects/BigInt](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/BigInt)
- [6] <http://ndjson.org/>
- [7] <http://ndjson.org/libraries.html>
- [8] <https://qlog.edm.uhasselt.be/anrw/>
- [9] <https://github.com/quiclog/internet-drafts/issues/30#issuecomment-617675097>
- [10] <https://www.chromium.org/developers/design-documents/network-stack/netlog>
- [11] <https://developers.google.com/protocol-buffers>
- [12] <https://github.com/quiclog/qlog>
- [13] <https://github.com/quiclog/qvis/tree/master/visualizations/src/components/filemanager/netlogconverter>
- [14] <https://github.com/quiclog/quictrace2qlog>
- [15] <https://qvis.edm.uhasselt.be>
- [16] <https://huitema.wordpress.com/2020/07/21/scrubbing-quic-logs-for-privacy/>
- [17] <https://github.com/google/quic-trace>
- [18] <https://github.com/EricssonResearch/spindump>
- [19] <https://www.wireshark.org/>

## Appendix A. Change Log

## A.1. Since draft-marx-qlog-main-schema-draft-02:

- o These changes were done in preparation of the adoption of the drafts by the QUIC working group (#137)
- o Moved RawInfo, Importance, Generic events and Simulation events to this document.
- o Added basic event definition guidelines
- o Made protocol\_type an array instead of a string (#146)

## A.2. Since draft-marx-qlog-main-schema-01:

- o Decoupled qlog from the JSON format and described a mapping instead (#89)
  - \* Data types are now specified in this document and proper definitions for fields were added in this format
  - \* 64-bit numbers can now be either strings or numbers, with a preference for numbers (#10)
  - \* binary blobs are now logged as lowercase hex strings (#39, #36)
  - \* added guidance to add length-specifiers for binary blobs (#102)
- o Removed "time\_units" from Configuration. All times are now in ms instead (#95)
- o Removed the "event\_fields" setup for a more straightforward JSON format (#101, #89)
- o Added a streaming option using the NDJSON format (#109, #2, #106)
- o Described optional optimization options for implementers (#30)
- o Added QLOGDIR and QLOGFILE environment variables, clarified the .well-known URL usage (#26, #33, #51)
- o Overall tightened up the text and added more examples

## A.3. Since draft-marx-qlog-main-schema-00:

- o All field names are now lowercase (e.g., category instead of CATEGORY)
- o Triggers are now properties on the "data" field value, instead of separate field types (#23)
- o group\_ids in common\_fields is now just also group\_id

## Appendix B. Design Variations

- o Quic-trace [17] takes a slightly different approach based on protocolbuffers.
- o Spindump [18] also defines a custom text-based format for in-network measurements
- o Wireshark [19] also has a QUIC dissector and its results can be transformed into a json output format using tshark.

The idea is that qlog is able to encompass the use cases for both of these alternate designs and that all tooling converges on the qlog standard.

## Appendix C. Acknowledgements

Much of the initial work by Robin Marx was done at Hasselt University.

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

## Authors' Addresses

Robin Marx (editor)  
KU Leuven

Email: robin.marx@kuleuven.be

Luca Niccolini (editor)  
Facebook

Email: lniccolini@fb.com

Marten Seemann (editor)  
Protocol Labs

Email: [marten@protocol.ai](mailto:marten@protocol.ai)