QUIC
Internet-Draft
Intended status: Standards Track
Expires: 23 April 2026

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

L. Pardue, Ed.
Cloudflare
20 October 2025

qlog: Structured Logging for Network Protocols
draft-ietf-quic-qlog-main-schema-13

Abstract

qlog provides extensible structured logging for network protocols, allowing for easy sharing of data that benefits common debug and analysis methods and tooling. This document describes key concepts of qlog: formats, files, traces, events, and extension points. This definition includes the high-level log file schemas, and generic event schemas. Requirements and guidelines for creating protocol-specific event schemas are also presented. All schemas are defined independent of serialization format, allowing logs to be represented in various ways such as JSON, CSV, or protobuf.

Note to Readers

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

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

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

Endpoint logging is a useful strategy for capturing and understanding how applications using network protocols are behaving, particularly where protocols have an encrypted wire image that restricts observers' ability to see what is happening.

Many applications implement logging using a custom, non-standard logging format. This has an effect on the tools and methods that are used to analyze the logs, for example to perform root cause analysis of an interoperability failure between distinct implementations. A lack of a common format impedes the development of common tooling that can be used by all parties that have access to logs.

qlog is an extensible structured logging for network protocols that allows for easy sharing of data that benefits common debug and analysis methods and tooling. This document describes key concepts of qlog: formats, files, traces, events, and extension points. This definition includes the high-level log file schemas, and generic event schemas. Requirements and guidelines for creating protocol-specific event schemas are also presented. Accompanying documents define event schemas for QUIC ([QLOG-QUIC]) and HTTP/3 ([QLOG-H3]).

The goal of qlog 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, qlog provides versioning, metadata inclusion, log aggregation, event grouping and log file size reduction techniques.

All qlog schemas can be serialized in many ways (e.g., JSON, CBOR, protobuf, etc). This document describes only how to employ [JSON], its subset [I-JSON], and its streamable derivative [JSON-Text-Sequences].

1.1. Conventions and Terminology

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

Serialization examples in this document use JSON ([JSON]) unless otherwise indicated.

Events are defined with an importance level as described in Section 8.3}.

1.2. Use of CDDL

To define events and data structures, all qlog documents use the Concise Data Definition Language [CDDL]. This document uses the basic syntax, the specific text, uint, float32, float64, bool, and any types, as well as the .default, .size, and .regexp control operators, the ~ unwrapping operator, and the $ and $$ extension points syntax from [CDDL].

Additionally, this document defines the following custom types for clarity:

```
; CDDL's uint is defined as being 64-bit in size
; but for many protocol fields it is better to be restrictive
; and explicit
uint8 = uint .size 1
uint16 = uint .size 2
uint32 = uint .size 4
uint64 = uint .size 8

; an even-length lowercase string of hexadecimally encoded bytes
; examples: 82dc, 027339, 4cdbfd9bf0
; this is needed because the default CDDL binary string (bytes/bstr)
; is only CBOR and not JSON compatible
hexstring = text .regexp "([0-9a-f]{2})*"
```

Figure 1: Additional CDDL type definitions

All timestamps and time-related values (e.g., offsets) in qlog are logged as float64 in the millisecond resolution.

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

The ordering of member fields in qlog CDDL type definitions is not significant. The ordering of member fields in the serialization formats defined in this document, JSON (Section 11.1) and JSON Text Sequences (Section 11.2), is not significant and qlog tools MUST NOT assume so. Other qlog serialization formats MAY define field order significance, if they do they MUST define requirements for qlog tools supporting those formats.

> Note to RFC editor: Please remove the following text in this section before publication.

The main general CDDL syntax conventions in this document a reader should be aware of for easy reading comprehension are:

* ? obj : this object is optional
* TypeName1 / TypeName2 : a union of these two types (object can be either type 1 OR type 2)
* obj: TypeName : this object has this concrete type
* obj: [* TypeName] : this object is an array of this type with minimum size of 0 elements
* obj: [+ TypeName] : this object is an array of this type with minimum size of 1 element
* TypeName = ... : defines a new type
* EnumName = "entry1" / "entry2" / entry3 / ...: defines an enum
* StructName = { ... } : defines a new struct type
* ; : single-line comment
* * text => any : special syntax to indicate 0 or more fields that have a string key that maps to any value. Used to indicate a generic JSON object.

All timestamps and time-related values (e.g., offsets) in qlog are logged as float64 in the millisecond resolution.

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

2. Design Overview

The main tenets for the qlog design are:

* Streamable, event-based logging
* A flexible format that can reduce log producer overhead, at the cost of increased complexity for consumers (e.g. tools)
* Extensible and pragmatic

* Aggregation and transformation friendly (e.g., the top-level element for the non-streaming format is a container for individual traces, group_ids can be used to tag events to a particular context)

* Metadata is stored together with event data

This is achieved by a logical logging hierarchy of:

* Log file

  - Trace(s)

    o Event(s)

An abstract LogFile class is declared (Section 3), from which all concrete log file formats derive using log file schemas. This document defines the QLogFile (Section 4) and QLogFileSeq (Section 5) log file schemas.

A trace is conceptually fluid but the conventional use case is to group events related to a single data flow, such as a single logical QUIC connection, at a single vantage point (Section 6). Concrete trace definitions relate to the log file schemas they are contained in; see (Section 4.1, Section 4.2, and Section 5.1).

Events are logged at a time instant and convey specific details of the logging use case. For example, a network packet being sent or received. This document declares an abstract Event class (Section 7) containing common fields, which all concrete events derive from. Concrete events are defined by event schemas that declare or extend a namespace, which contains one or more related event types or their extensions. For example, this document defines two event schemas for two generic event namespaces loglevel and simulation (see Section 9).

3. Abstract LogFile Class

A Log file is intended to contain a collection of events that are in some way related. An abstract LogFile class containing fields common to all log files is defined in Figure 2. Each concrete log file schema derives from this using the CDDL unwrap operator (~) and can extend it by defining semantics and any custom fields.

```
LogFile = {
    file_schema: text
    serialization_format: text
    ? title: text
    ? description: text
}
```

Figure 2: LogFile definition

The required "file_schema" field identifies the concrete log file schema. It MUST have a value that is an absolute URI; see Section 3.1 for rules and guidance.

The required "serialization_format" field indicates the serialization format using a media type [RFC2046]. It is case-insensitive.

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

The optional "title" and "description" fields provide additional free-text information about the file.

3.1. Concrete Log File Schema URIs

Concrete log file schemas MUST identify themselves using a URI [RFC3986].

Log file schemas defined by RFCs MUST register a URI in the "qlog log file schema URIs" registry and SHOULD use a URN of the form urn:ietf:params:qlog:file:<schema-identifier>, where <schema-identifier> is a globally-unique text name using only characters in the URI unreserved range; see Section 2.3 of [RFC3986]. This document registers urn:ietf:params:qlog:file:contained (Section 4) and urn:ietf:params:qlog:file:sequential (Section 5).

Private or non-standard log file schemas MAY register a URI in the "qlog log file schema URIs" registry but MUST NOT use a URN of the form urn:ietf:params:qlog:file:<schema-identifier>. URIs that contain a domain name SHOULD also contain a month-date in the form mmyyyy. For example, "https://example.org/072024/globallyuniquelogfileschema". The definition of the log file schema and assignment of the URI MUST have been authorized by the owner of the domain name on or very close to that date. This avoids problems when domain names change ownership. The URI does not need to be dereferencable, allowing for confidential use or to cover the case where the log file schema continues to be used after the organization that defined them ceases to exist.

The "qlog log file schema URIs" registry operates under the Expert Review policy, per Section 4.5 of [RFC8126]. When reviewing requests, the expert MUST check that the URI is appropriate to the concrete log file schema and satisfies the requirements in this section. A request to register a private or non-standard log file schema URI using a URN of the form urn:ietf:params:qlog:file:<schema-identifier> MUST be rejected.

Registration requests should use the template defined in Section 15.

4. QlogFile schema

A qlog file using the QlogFile schema can contain several individual traces and logs from multiple vantage points that are in some way related. The top-level element in this schema defines only a small set of "header" fields and an array of component traces. This is defined in Figure 3 as:

```
QlogFile = {
    ~LogFile
    ? traces: [+ Trace /
                 TraceError]
}
```

Figure 3: QlogFile definition

The QlogFile schema URI is urn:ietf:params:qlog:file:contained.

QlogFile extends LogFile using the CDDL unwrap operator (~), which copies the fields presented in Section 3. Additionally, the optional "traces" field contains an array of qlog traces (Section 4.2), each of which contain metadata and an array of qlog events (Section 7).

The default serialization format for QlogFile is JSON; see Section 11.1 for guidance on populating the "serialization_format" field and other considerations. Where a qlog file is serialized to a JSON format, one of the downsides 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. The alternative QlogFileSeq (Section 5) is better suited to streaming use cases.

JSON serialization example:

```
{
    "file_schema": "urn:ietf:params:qlog:file:contained",
    "serialization_format": "application/qlog+json",
    "title": "Name of this particular qlog file (short)",
    "description": "Description for this group of traces (long)",
    "traces": [...]
}
```

Figure 4: QlogFile example

4.1. Traces

It can be advantageous to group several related qlog traces together in a single file. For example, it is possible to 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.

The QlogFile "traces" field is an array that contains a list of individual qlog traces. When capturing a qlog at a vantage point, it is expected that the traces field contains a single entry. Files can be aggregated, for example as part of a post-processing operation, by copying the traces in component to files into the combined "traces" array of a new, aggregated qlog file.

4.2. Trace

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. 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.

A Trace contains some metadata in addition to qlog events, defined in Figure 5 as:

```
Trace = {
    ? title: text
    ? description: text
    ? common_fields: CommonFields
    ? vantage_point: VantagePoint
    event_schemas: [+text]
    events: [* Event]
}
```

Figure 5: Trace definition

The optional "title" and "description" fields provide additional free-text information about the trace.

The optional "common_fields" field is described in Section 7.5.

The optional "vantage_point" field is described in Section 6.

The required "event_schemas" field contains event schema URIs that identify concrete event namespaces and their associated types recorded in the "events" field. Requirements and guidelines are defined in Section 8.

The semantics and context of the trace can mainly be deduced from the entries in the "common_fields" list and "vantage_point" field.

JSON serialization example:

```
{
    "title": "Name of this particular trace (short)",
    "description": "Description for this trace (long)",
    "common_fields": {
        "ODCID": "abcde1234",
        "time_format": "relative_to_epoch",
        "reference_time": {
            "clock_type": "system",
            "epoch": "1970-01-01T00:00:00.000Z"
        },
    },
    "vantage_point": {
        "name": "backend-67",
        "type": "server"
    },
    "event_schemas": ["urn:ietf:params:qlog:events:quic"],
    "events": [...]
}
```

Figure 6: Trace example

4.3. TraceError

A TraceError indicates that an attempt to find/convert a file for inclusion in the aggregated qlog was made, but there was an error during the process. Rather than silently dropping the erroneous file, it can be explicitly included in the qlog file as an entry in the "traces" array, defined in Figure 7 as:

```
TraceError = {
    error_description: text

    ; the original URI used for attempted find of the file
    ? uri: text
    ? vantage_point: VantagePoint
}
```

Figure 7: TraceError definition

JSON serialization example:

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

Figure 8: TraceError example

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 7.3.

5. QlogFileSeq schema

A qlog file using the QlogFileSeq schema can be serialized to a streamable JSON format called JSON Text Sequences (JSON-SEQ) ([RFC7464]). The top-level element in this schema defines only a small set of "header" fields and an array of component traces. This is defined in Figure 3 as:

```
QlogFileSeq = {
    ~LogFile
    trace: TraceSeq
}
```

Figure 9: QlogFileSeq definition

The QlogFileSeq schema URI is urn:ietf:params:qlog:file:sequential.

QlogFile extends LogFile using the CDDL unwrap operator (~), which copies the fields presented in Section 3. Additionally, the required "trace" field contains a singular trace (Section 4.2). All qlog events in the file are related to this trace; see Section 5.1.

See Section 11.2 for guidance on populating the "serialization_format" field and other serialization considerations.

JSON-SEQ serialization example:

```
// list of qlog events, serialized in accordance with RFC 7464,
// starting with a Record Separator character and ending with a
// newline.
// For display purposes, Record Separators are rendered as <RS>

<RS>{
    "file_schema": "urn:ietf:params:qlog:file:sequential",
    "serialization_format": "application/qlog+json-seq",
    "title": "Name of JSON Text Sequence qlog file (short)",
    "description": "Description for this trace file (long)",
    "trace": {
      "common_fields": {
        "group_id":"127ecc830d98f9d54a42c4f0842aa87e181a",
        "time_format": "relative_to_epoch",
        "reference_time": {
            "clock_type": "system",
            "epoch": "1970-01-01T00:00:00.000Z"
        },
      },
      "vantage_point": {
        "name":"backend-67",
        "type":"server"
      },
      "event_schemas": ["urn:ietf:params:qlog:events:quic",
                        "urn:ietf:params:qlog:events:http3"]
    }
}
<RS>{"time": 2, "name": "quic:parameters_set", "data": { ... } }
<RS>{"time": 7, "name": "quic:packet_sent", "data": { ... } }
...
```

Figure 10: Top-level element

5.1. TraceSeq

TraceSeq is used with QlogFileSeq. It is conceptually similar to a Trace, with the exception that qlog events are not contained within it, but rather appended after it in a QlogFileSeq.

```
TraceSeq = {
    ? title: text
    ? description: text
    ? common_fields: CommonFields
    ? vantage_point: VantagePoint
    event_schemas: [+text]
}
```

Figure 11: TraceSeq definition

6. VantagePoint

A VantagePoint describes the vantage point from which a trace originates, defined in Figure 12 as:

```
VantagePoint = {
    ? name: text
    type: VantagePointType
    ? flow: VantagePointType
}

; client = endpoint which initiates the connection
; server = endpoint which accepts the connection
; network = observer in between client and server
VantagePointType = "client" /
                   "server" /
                   "network" /
                   "unknown"
```

Figure 12: VantagePoint definition

JSON serialization examples:

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

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

Figure 13: VantagePoint example

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.

7. Abstract Event Class

Events are logged at a time instant and convey specific details of the logging use case. An abstract Event class containing fields common to all events is defined in Figure 14.

```
Event = {
    time: float64
    name: text
    data: $ProtocolEventData
    ? tuple: TupleID
    ? time_format: TimeFormat
    ? group_id: GroupID
    ? system_info: SystemInformation

    ; events can contain any amount of custom fields
    * text => any
}
```

Figure 14: Event definition

Each qlog event MUST contain the mandatory fields: "time" (Section 7.1), "name" (Section 8), and "data" (Section 8.2).

Each qlog event is an instance of a concrete event type that derives from the abstract Event class; see Section 8. They extend it by defining the specific values and semantics of common fields, in particular the name and data fields. Furthermore, they can optionally add custom fields.

Each qlog event MAY contain the optional fields: "time_format" (Section 7.1), tuple (Section 7.2) "trigger" (Section 8.2.1), and "group_id" (Section 7.3).

Multiple events can appear in a Trace or TraceSeq and they might contain fields with identical values. It is possible to optimize out this duplication using "common_fields" (Section 7.5).

Example qlog event:

```
{
    "time": 1553986553572,

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

    "group_id": "127ecc830d98f9d54a42c4f0842aa87e181a",

    "time_format": "relative_to_epoch",

    "ODCID": "127ecc830d98f9d54a42c4f0842aa87e181a"
}
```

Figure 15: Event example

7.1. Timestamps

Each event MUST include a "time" field to indicate the timestamp that it occurred. It is a duration measured from some point in time; its units depend on the type of clock chosen and system used. The time field is a float64 and it is typically used to represent a duration in milliseconds, with a fractional component to microsecond or nanosecond resolution.

There are several options for generating and logging timestamps, these are governed by the ReferenceTime type (optionally included in the "reference_time" field contained in a trace's "common_fields" (Section 7.5)) and TimeFormat type (optionally included in the "time_format" field contained in the event itself, or a trace's "common_fields").

There is no requirement that events in the same trace use the same time format. However, using a single time format for related events can make them easier to analyze.

The reference time governs from which point in time the "time" field values are measured and is defined as:

```
ReferenceTime = {
    clock_type: "system" / "monotonic" / text .default "system"
    epoch: RFC3339DateTime / "unknown" .default "1970-01-01T00:00:00.000Z"

    ? wall_clock_time: RFC3339DateTime
}

RFC3339DateTime = text
```

Figure 16: ReferenceTime definition

The required "clock_type" field represents the type of clock used for time measurements. The value "system" represents a clock that uses system time, commonly measured against a chosen or well-known epoch. However, depending on the system, System time can potentially jump forward or back. In contrast, a clock using monotonic time is generally guaranteed to never go backwards. The value "monotonic" represents such a clock.

The required "epoch" field is the start of the ReferenceTime. When using the "system" clock type, the epoch field SHOULD have a date/time value using the format defined in [RFC3339]. However, the value "unknown" MAY be used.

When using the "monotonic" clock type, the epoch field MUST have the value "unknown".

The optional "wall_clock_time" field can be used to provide an approximate date/time value that logging commenced at if the epoch value is "unknown". It uses the format defined in [RFC3339]. Note that conversion of timestamps to calendar time based on wall clock times cannot be safely relied on.

The time format details how "time" values are encoded relative to the reference time and is defined as:

```
TimeFormat = "relative_to_epoch" /
             "relative_to_previous_event" .default "relative_to_epoch"
```

Figure 17: TimeFormat definition

relative_to_epoch: A duration relative to the ReferenceTime "epoch" field. This approach uses the largest amount of characters. It is good for stateless loggers. This is the default value of the "time_format" field.

relative_to_previous_event: A delta-encoded value, based on the

previously logged value. The first event in a trace is always relative to the ReferenceTime. This approach uses the least amount of characters. It is suitable for stateful loggers.

Events in each individual trace SHOULD be logged in strictly ascending timestamp order (though not necessarily absolute value, for the "relative_to_previous_event" format). Tools MAY 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 post-processor to order qlog events in time if a tool do not provide this feature.

Tools SHOULD NOT assume the ability to derive the absolute calendar timestamp of an event from qlog traces. Tools should not rely on timestamps to be consistent across traces, even those generated by the same logging endpoint. For reasons of privacy, the reference time MAY have minimization or anonymization applied.

Example of a log using the relative_to_epoch format:

```
   "common_fields": {
       "time_format": "relative_to_epoch",
       "reference_time": {
             "clock_type": "system",
             "epoch": "1970-01-01T00:00:00.000Z"
       },
   },
   "events": [
     {
       "time": 1553986553572,
       "name": "quic:packet_received",
       "data": { ... },
     },
     {
       "time": 1553986553577,
       "name": "quic:packet_received",
       "data": { ... },
     },
     {
       "time": 1553986553587,
       "name": "quic:packet_received",
       "data": { ... },
     },
     {
       "time": 1553986553597,
       "name": "quic:packet_received",
       "data": { ... },
     },
   ]
```

Figure 18: Relative to epoch timestamps

Example of a log using the relative_to_previous_event format:

```
   "common_fields": {
       "time_format": "relative_to_previous_event",
       "reference_time": {
             "clock_type": "system",
             "epoch": "1970-01-01T00:00:00.000Z"
       },
   },
   "events": [
     {
       "time": 1553986553572,
       "name": "quic:packet_received",
       "data": { ... },
     },
     {
       "time": 5,
       "name": "quic:packet_received",
       "data": { ... },
     },
     {
       "time": 10,
       "name": "quic:packet_received",
       "data": { ... },
     },
     {
       "time": 10,
       "name": "quic:packet_received",
       "data": { ... },
     },
   ]
```

Figure 19: Relative-to-previous-event timestamps

Example of a monotonic log using the relative_to_epoch format:

```
   "common_fields": {
       "time_format": "relative_to_epoch",
       "reference_time": {
             "clock_type": "monotonic",
             "epoch": "unknown",
             "wall_clock_time": "2024-10-10T10:10:10.000Z"
       },
   },
   "events": [
     {
       "time": 0,
       "name": "quic:packet_received",
       "data": { ... },
     },
     {
       "time": 5,
       "name": "quic:packet_received",
       "data": { ... },
     },
     {
       "time": 15,
       "name": "quic:packet_received",
       "data": { ... },
     },
     {
       "time": 25,
       "name": "quic:packet_received",
       "data": { ... },
     },
   ]
```

Figure 20: Monotonic timestamps

7.2. Tuple

A qlog event is typically associated with a single network "path", which is usually aligned with a four-tuple of IP addresses and ports. In many cases, this tuple will be the same for all events in a given trace, and does not need to be logged explicitly with each event. In this case, the "tuple" field can be omitted (in which case the default value of "" is assumed) or reflected in "common_fields" instead (see Section 7.5).

However, in some situations, such as during QUIC’s Connection Migration or when using Multipath features, it is useful to be able to split events across multiple (concurrent) tuples and/or paths.

Definition:

```
TupleID = text .default ""
```

Figure 21: TupleID definition

The "tuple" field is an identifier that is associated with a single network four-tuple. This document intentionally does not define further how to choose this identifier's value per-tuple or how to potentially log other parameters that can be associated with such a tuple. This is left for other documents. Implementers are free to encode tuple information directly into the TupleID or to log associated info in a separate event. For example, QUIC has the "tuple_assigned" event to couple the TupleID value to a specific tuple configuration, see [QLOG-QUIC].

7.3. Grouping

As discussed in Section 4.2, 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, a method for splitting up events belonging to separate logical entities is required.

The simplest way to perform this splitting is by associating a "group id" 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, the QUIC "Original Destination Connection ID" could be used to uniquely identify a connection. As such, they might add a "ODCID" field to each event. Additionally, a service providing different levels of Quality of Service (QoS) to their users might wish to group connections per QoS level applied. They might instead prefer a "qos" 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 23.

```
GroupID = text
```

Figure 22: GroupID definition

JSON serialization example for events grouped either by QUIC Connection IDs, or according to an endpoint-specific Quality of Service (QoS) logic that includes the service level:

```
"events": [
    {
        "time": 1553986553579,
        "group_id": "qos=premium",
        "name": "quic:packet_received",
        "data": { ... }
    },
    {
        "time": 1553986553581,
        "group_id": "127ecc830d98f9d54a42c4f0842aa87e181a",
        "name": "quic:packet_sent",
        "data": { ... }
    }
]
```

Figure 23: GroupID example

Note that in some contexts (for example a Multipath transport protocol) it might make sense to add additional contextual per-event fields (for example TupleID, see Section 7.2), 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 7.5.

7.4. SystemInformation

The "system_info" field can be used to record system-specific details related to an event. This is useful, for instance, where an application splits work across CPUs, processes, or threads and events for a single trace occur on potentially different combinations thereof. Each field is optional to support deployment diversity.

```
SystemInformation = {
  ? processor_id: uint32
  ? process_id: uint32
  ? thread_id: uint32
}
```

7.5. CommonFields

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 "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 24.

JSON serialization with repeated field values per-event instance:

```
{
    "events": [{
            "group_id": "127ecc830d98f9d54a42c4f0842aa87e181a",
            "time_format": "relative_to_epoch",
            "reference_time": {
              "clock_type": "system",
              "epoch": "2019-03-29T:22:55:53.572Z"
            },

            "time": 2,
            "name": "quic:packet_received",
            "data": { ... }
        },{
            "group_id": "127ecc830d98f9d54a42c4f0842aa87e181a",
            "time_format": "relative_to_epoch",
            "reference_time": {
              "clock_type": "system",
              "epoch": "2019-03-29T:22:55:53.572Z"
            },

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

JSON serialization with repeated field values instead

extracted to common_fields:

```
{
    "common_fields": {
        "group_id": "127ecc830d98f9d54a42c4f0842aa87e181a",
        "time_format": "relative_to_epoch",
        "reference_time": {
            "clock_type": "system",
            "epoch": "2019-03-29T:22:55:53.572Z"
        },
    },
    "events": [
        {
            "time": 2,
            "name": "quic:packet_received",
            "data": { ... }
        },{
            "time": 7,
            "name": "http:frame_parsed",
            "data": { ... }
        }
    ]
}
```

Figure 24: CommonFields example

An event's "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 shown in the listing below:

```
CommonFields = {
    ? tuple: TupleID
    ? time_format: TimeFormat
    ? reference_time: ReferenceTime
    ? group_id: GroupID
    * text => any
}
```

Figure 25: CommonFields definition

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.

8. Concrete Event Types and Event Schemas

Concrete event types, as well as related data types, are grouped in event namespaces which in turn are defined in one or multiple event schemas.

As an example, the QUICPacketSent and QUICPacketHeader event and data types would be part of the quic namespace, which is defined in an event schema with URI urn:ietf:params:qlog:events:quic. A later extension that adds a new QUIC frame QUICNewFrame would also be part of the quic namespace, but defined in a new event schema with URI urn:ietf:params:qlog:events:quic#new-frame-extension.

Concrete event types MUST belong to a single event namespace and MUST have a registered non-empty identifier of type text.

New namespaces MUST have a registered non-empty globally-unique text identifier using only characters in the URI unreserved range; see Section 2.3 of [RFC3986]. Namespaces are mutable and MAY be extended with new events.

The value of a qlog event name field MUST be the concatenation of namespace identifier, colon (':'), and event type identifier (for example: quic:packet_sent). The resulting concatenation MUST be globally unique, so log files can contain events from multiple event schemas without the risk of name collisions.

A single event schema can contain exactly one of the below:

* A definition for a new event namespace
* An extension of an existing namespace (adding new events/data types and/or extending existing events/data types within the namespace with new fields)

A single document can define multiple event schemas (for example see Section 9).

An event schema MUST have a single URI [RFC3986] that MUST be absolute. The URI MUST include the namespace identifier. Event schemas that extend an existing namespace MUST furthermore include a

non-empty globally-unique "extension" identifier using a URI fragment (characters after a "#" in the URI) using only characters in the URI unreserved range; see Section 2.3 of [RFC3986]. Registration guidance and requirement for event schema URIs are provided in Section 8.1. Event schemas by themselves are immutable and MUST NOT be extended.

Implementations that record concrete event types SHOULD list all event schemas in use. This is achieved by including the appropriate URIs in the event_schemas field of the Trace (Section 4.2) and TraceSeq (Section 5.1) classes. The event_schemas is a hint to tools about the possible event namespaces, their extensions, and the event types/data types contained therein, that a qlog trace might contain. The trace MAY still contain event types that do not belong to a listed event schema. Inversely, not all event types associated with an event schema listed in event_schemas are guaranteed to be logged in a qlog trace. Tools MUST NOT treat either of these as an error; see Section 13.

In the following hypothetical example, a qlog trace contains events belonging to:

* The two event namespaces defined by event schemas in this document (Section 9).

* Events in a namespace named rick specified in a hypothetical RFC

* Extentions to the rick namespace defined in two separate new event schemas (with URI extension identifiers astley and moranis)

* Events from three private event schemas, detailing definitions for and extensions to two namespaces (pickle and cucumber)

The standardized schema URIs use a URN format, the private schemas use a URI with domain name.

```
"event_schemas": [
  "urn:ietf:params:qlog:events:loglevel",
  "urn:ietf:params:qlog:events:simulation",
  "urn:ietf:params:qlog:events:rick",
  "urn:ietf:params:qlog:events:rick#astley",
  "urn:ietf:params:qlog:events:rick#moranis",
  "https://example.com/032024/pickle.html",
  "https://example.com/032024/pickle.html#lilly",
  "https://example.com/032025/cucumber.html"
]
```

Figure 26: Example event_schemas serialization

8.1. Event Schema URIs

Event schemas defined by RFCs MUST register all namespaces and concrete event types they contain in the "qlog event schema URIs" registry.

Event schemas that define a new namespace SHOULD use a URN of the form urn:ietf:params:qlog:events:<namespace identifier>, where <namespace identifier> is globally unique. For example, this document defines two event schemas (Section 9) for two namespaces: loglevel and sim. Other examples of event schema define the quic [QLOG-QUIC] and http3 [QLOG-H3] namespaces.

Event schemas that extend an existing namespace SHOULD use a URN of the form urn:ietf:params:qlog:events:<namespace identifier>#<extension identifier>, where the combination of <namespace identifier> and <extension identifier> is globally unique.

Private or non-standard event schemas MAY be registered in the "qlog event schema URIs" registry but MUST NOT use a URN of the forms outlined above. URIs that contain a domain name SHOULD also contain a month-date in the form mmyyyy. For example, "https://example.org/072024/customeventschema#customextension". The definition of the event schema and assignment of the URI MUST have been authorized by the owner of the domain name on or very close to that date. This avoids problems when domain names change ownership. The URI does not need to be dereferencable, allowing for confidential use or to cover the case where the event schemas continue to be used after the organization that defined them ceases to exist.

The "qlog event schema URIs" registry operates under the Expert Review policy, per Section 4.5 of [RFC8126]. When reviewing requests, the expert MUST check that the URI is appropriate to the event schema and satisfies the requirements in Section 8 and this section. A request to register a private or non-standard schema URI using a URN of the forms reserved for schemas defined by an RFC above MUST be rejected.

Registration requests should use the template defined in Section 15.

8.2. Extending the Data Field

An event’s "data" field is a generic key-value map (e.g., JSON object). It defines the per-event metadata that is to be logged. Its specific subfields and their semantics are defined per concrete event type. For example, data field definitions for QUIC and HTTP/3 can be found in [QLOG-QUIC] and [QLOG-H3].

In order to keep qlog fully extensible, two separate CDDL extension points ("sockets" or "plugs") are used to fully define data fields.

Firstly, to allow existing data field definitions to be extended (for example by adding an additional field needed for a new protocol feature), a CDDL "group socket" is used. This takes the form of a subfield with a name of * $$NAMESPACE-EVENTTYPE-extension. This field acts as a placeholder that can later be replaced with newly defined fields by assigning them to the socket with the //= operator. Multiple extensions can be assigned to the same group socket. An example is shown in Figure 27.

```
; original definition in event schema A
MyNSEventX = {
    field_a: uint8

    * $$myns-eventx-extension
}

; later extension of EventX in event schema B
$$myns-eventx-extension //= (
  ? additional_field_b: bool
)

; another extension of EventX in event schema C
$$myns-eventx-extension //= (
  ? additional_field_c: text
)

; if schemas A, B and C are then used in conjunction,
; the combined MyNSEventX CDDL is equivalent to this:
MyNSEventX = {
    field_a: uint8

    ? additional_field_b: bool
    ? additional_field_c: text
}
```

Figure 27: Example of using a generic CDDL group socket to extend an existing event data definition

Secondly, to allow documents to define fully new event data field definitions (as opposed to extend existing ones), a CDDL "type socket" is used. For this purpose, the type of the "data" field in the qlog Event type (see Figure 14) is the extensible $ProtocolEventData type. This field acts as an open enum of possible types that are allowed for the data field. As such, any new event data field is defined as its own CDDL type and later merged with the

existing $ProtocolEventData enum using the /= extension operator. Any generic key-value map type can be assigned to $ProtocolEventData. The example in Figure 28 demonstrates $ProtocolEventData being extended with two types.

```
; We define two new concrete events in a new event schema
MyNSEvent1 /= {
    field_1: uint8

    * $$myns-event1-extension
}

MyNSEvent2 /= {
    field_2: bool

    * $$myns-event2-extension
}

; the events are both merged with the existing
; $ProtocolEventData type enum
$ProtocolEventData /= MyNSEvent1 / MyNSEvent2

; the "data" field of a qlog event can now also be of type
; MyNSEvent1 and MyNSEvent2
```

Figure 28: ProtocolEventData extension

Event schema defining new qlog events MUST properly extend $ProtocolEventData when defining data fields to enable automated validation of aggregated qlog schemas. Furthermore, they SHOULD add a * $$NAMESPACE-EVENTTYPE-extension extension field to newly defined event data to allow the new events to be properly extended by other event schema.

A combined but purely illustrative example of the use of both extension points for a conceptual QUIC "packet_sent" event is shown in Figure 29:

```
; defined in the main QUIC event schema
QUICPacketSent = {
    ? packet_size: uint16
    header: QUICPacketHeader
    ? frames:[* QUICFrame]

    * $$quic-packetsent-extension
}

; Add the event to the global list of recognized qlog events
$ProtocolEventData /= QUICPacketSent

; Defined in a separate event schema that describes a
; theoretical QUIC protocol extension
$$quic-packetsent-extension //= (
  ? additional_field: bool
)

; If both schemas are utilized at the same time,
; the following JSON serialization would pass an automated
; CDDL schema validation check:

{
  "time": 123456,
  "name": "quic:packet_sent",
  "data": {
      "packet_size": 1280,
      "header": {
          "packet_type": "1RTT",
          "packet_number": 123
      },
      "frames": [
          {
              "frame_type": "stream",
              "offset": 456
          },
          {
              "frame_type": "padding"
          }
      ],
      additional_field: true
  }
}
```

Figure 29: Example of an extended ’data’ field for a conceptual QUIC packet_sent event

8.2.1. Triggers

It can be useful to understand the cause or trigger of an event. Sometimes, events are caused by a variety of other events and additional information is needed to identify the exact details. Commonly, the context of the surrounding log messages gives a hint about the cause. However, in highly-parallel and optimized implementations, corresponding log messages might be separated in time, making it difficult to build an accurate context.

Including a "trigger" as part of the event itself is one method for providing fine-grained information without much additional overhead. In circumstances where a trigger is useful, it is RECOMMENDED for the purpose of consistency that the event data definition contains an optional field named "trigger", holding a string value.

For example, the QUIC "packet_dropped" event (Section 5.7 of [QLOG-QUIC]) includes a trigger field that identifies the precise reason why a QUIC packet was dropped:

```
QUICPacketDropped = {

    ; Primarily packet_type should be filled here,
    ; as other fields might not be decrypteable or parseable
    ? header: PacketHeader
    ? raw: RawInfo
    ? datagram_id: uint32
    ? details: {* text => any}
    ? trigger:
        "internal_error" /
        "rejected" /
        "unsupported" /
        "invalid" /
        "duplicate" /
        "connection_unknown" /
        "decryption_failure" /
        "key_unavailable" /
        "general"

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

Figure 30: Trigger example

8.3. Event Importance Levels

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, qlog defines three event importance levels, in decreasing order of importance and expected usage:

* Core
* Base
* Extra

Concrete event types SHOULD define an importance level.

Core-level events 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.

Base-level events add additional debugging options and MAY 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 used to 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.

Extra-level events are considered mostly useful for low-level debugging of the implementation, rather than the protocol. They allow more fine-grained tracking of internal behavior. As such, they MAY be present in qlog files and tool implementers MAY 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 Core-level events due to performance or privacy considerations. In this case, they SHOULD use (a subset of) relevant Base-level 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.

8.4. Tooling Expectations

qlog is an extensible format and it is expected that new event schema will emerge that define new namespaces, event types, event fields (e.g., a field indicating an event's privacy properties), as well as values for the "trigger" property within the "data" field, or other member fields of the "data" field, as they see fit.

It SHOULD NOT be expected that general-purpose tools will recognize or visualize all forms of qlog extension. Tools SHOULD allow for the presence of unknown event fields and make an effort to visualize even unknown data if possible, otherwise they MUST ignore it.

8.5. Further Design Guidance

There are several ways of defining concrete event types. In practice, two main types of approach have been observed: 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 behaviors 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, when logging additional state change events, those state changes can often be directly inferred from data on the wire (for example flow control limit changes). As such, if the implementation is bug-free and spec-compliant, logging additional events is typically avoided. Exceptions have been made for common events that benefit from being easily identifiable or individually logged (for example packets_acked).

9. The Generic Event Schemas

The two following generic event schemas define two namespaces and several concrete event types that are common across protocols, applications, and use cases.

9.1. Loglevel events

In typical logging setups, users utilize a discrete number of well-defined logging categories, levels or severities to log freeform (string) data. The loglevel event namespace 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). The namespace identifier is "loglevel". The event schema URI is urn:ietf:params:qlog:events:loglevel.

```
LogLevelEventData = LogLevelError /
                    LogLevelWarning /
                    LogLevelInfo /
                    LogLevelDebug /
                    LogLevelVerbose

$ProtocolEventData /= LogLevelEventData
```

Figure 31: LogLevelEventData and ProtocolEventData extension

The event types are further defined below, their identifier is the heading name.

9.1.1. error

Used to log details of an internal error that might not get reflected on the wire. It has Core importance level.

```
LogLevelError = {
    ? code: uint64
    ? message: text

    * $$loglevel-error-extension
}
```

Figure 32: LogLevelError definition

9.1.2. warning

Used to log details of an internal warning that might not get reflected on the wire. It has Base importance level.

```
LogLevelWarning = {
    ? code: uint64
    ? message: text

    * $$loglevel-warning-extension
}
```

Figure 33: LogLevelWarning definition

9.1.3. info

Used mainly for implementations that want to use qlog as their one and only logging format but still want to support unstructured string messages. The event has Extra importance level.

```
LogLevelInfo = {
    message: text

    * $$loglevel-info-extension
}
```

Figure 34: LogLevelInfo definition

9.1.4. debug

Used mainly for implementations that want to use qlog as their one and only logging format but still want to support unstructured string messages. The event has Extra importance level.

```
LogLevelDebug = {
    message: text

    * $$loglevel-debug-extension
}
```

Figure 35: LogLevelDebug definition

9.1.5. verbose

Used mainly for implementations that want to use qlog as their one and only logging format but still want to support unstructured string messages. The event has Extra importance level.

```
LogLevelVerbose = {
    message: text

    * $$loglevel-verbose-extension
}
```

Figure 36: LogLevelVerbose definition

9.2. 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. This namespace defines event types to allow logging of such simulation metadata and its identifier is "simulation". The event schema URI is urn:ietf:params:qlog:events:simulation.

```
SimulationEventData = SimulationScenario /
                      SimulationMarker

$ProtocolEventData /= SimulationEventData
```

Figure 37: SimulationEventData and ProtocolEventData extension

The event types are further defined below, their identifier is the heading name.

9.2.1. scenario

Used to specify which specific scenario is being tested at this particular instance. This supports, for example, aggregation of several simulations into one trace (e.g., split by group_id). It has Extra importance level; see Section 8.3.

```
SimulationScenario = {
    ? name: text
    ? details: {* text => any }

    * $$simulation-scenario-extension
}
```

Figure 38: SimulationScenario definition

9.2.2. marker

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). It has Extra importance level.

```
SimulationMarker = {
    ? type: text
    ? message: text

    * $$simulation-marker-extension
}
```

Figure 39: SimulationMarker definition

10. Raw packet and frame information

While qlog is a high-level logging format, it also allows the inclusion of most raw wire image information, such as byte lengths and byte values. This is useful when for example investigating or tuning packetization behavior or determining encoding/framing overheads. However, these fields are not always necessary, can take up considerable space, and can have a considerable privacy and security impact (see Section 14). Where applicable, these fields are grouped in a separate, optional, field named "raw" of type RawInfo. The exact definition of entities, headers, trailers and payloads depend on the protocol used.

```
RawInfo = {

    ; the full byte length of the entity (e.g., packet or frame),
    ; including possible headers and trailers
    ? length: uint64

    ; the byte length of the entity's payload,
    ; excluding possible headers or trailers
    ? payload_length: uint64

    ; the (potentially truncated) contents of the full entity,
    ; including headers and possibly trailers
    ? data: hexstring
}
```

Figure 40: RawInfo definition

All fields in RawInfo are defined as optional. It is acceptable to log any field without the others. Logging length related fields and omitting the data field permits protocol debugging without the risk of logging potentially sensitive data. The data field, if logged, is not required to contain the contents of a full entity and can be truncated, see Section 11.4. The length fields, if logged, should indicate the length of the the full entity, even if the data field is omitted or truncated.

Protocol entities containing an on-the-wire length field (for example a packet header or QUIC's stream frame) are strongly recommended to re-use the raw.length field instead of defining a separate length field, to maintain consistency and prevent data duplication.

This document does not specify explicit header_length or trailer_length fields. In protocols without trailers, header_length can be calculated by subtracting the payload_length from the length. In protocols with trailers (e.g., QUIC's AEAD tag), event definition documents SHOULD define how to support header_length calculation.

11. Serializing qlog

qlog schema definitions in this document are intentionally agnostic to serialization formats. The choice of format is an implementation decision.

Other documents related to qlog (for example event definitions for specific protocols), SHOULD be similarly agnostic to the employed serialization format and SHOULD clearly indicate this. If not, they MUST include an explanation on which serialization formats are supported and on how to employ them correctly.

Serialization formats make certain tradeoffs between usability, flexibility, interoperability, and efficiency. Implementations should take these into consideration when choosing a format. Some examples of possible formats are JSON, CBOR, CSV, protocol buffers, flatbuffers, etc. which each have their own characteristics. For instance, a textual format like JSON can be more flexible than a binary format but more verbose, typically making it less efficient than a binary format. A plaintext readable (yet relatively large) format like JSON is potentially more usable for users operating on the logs directly, while a more optimized yet restricted format can better suit the constraints of a large scale operation. A custom or restricted format could be more efficient for analysis with custom tooling but might not be interoperable with general-purpose qlog tools.

Considering these tradeoffs, JSON-based serialization formats provide features that make them a good starting point for qlog flexibility and interoperability. For these reasons, JSON is a recommended default and expanded considerations are given to how to map qlog to JSON (Section 11.1, and its streaming counterpart JSON Text Sequences (Section 11.2. Section 11.3 presents interoperability considerations for both formats, and Section 11.5 presents potential optimizations.

Serialization formats require appropriate deserializers/parsers. The "serialization_format" field (Section 3) is used to indicate the chosen serialization format.

11.1. qlog to JSON mapping

As described in Section 4, JSON is the default qlog serialization. When mapping qlog to normal JSON, QlogFile (Figure 3) is used. The Media Type is "application/qlog+json" per [RFC6839]. The file extension/suffix SHOULD be ".qlog".

In accordance with Section 8.1 of [RFC8259], JSON files are required to use UTF-8 both for the file itself and the string values it contains. In addition, all qlog field names MUST be lowercase when serialized to JSON.

In order to serialize CDDL-based qlog event and data structure definitions to JSON, the official CDDL-to-JSON mapping defined in Appendix E of [CDDL] SHOULD be employed.

11.2. qlog to JSON Text Sequences mapping

One of the downsides of using normal JSON is that it is inherently a non-streamable format. A qlog serializer could work around this by opening a file, writing the required opening data, streaming qlog events by appending them, and then finalizing the log by appending appropriate closing tags e.g., "]}]}". However, failure to append closing tags, could lead to problems because most JSON parsers will fail if a document is malformed. Some streaming JSON parsers are able to handle missing closing tags, however they are not widely deployed in popular environments (e.g., Web browsers)

To overcome the issues related to JSON streaming, a qlog mapping to a streamable JSON format called JSON Text Sequences (JSON-SEQ) ([RFC7464]) is provided.

JSON Text Sequences are very similar to JSON, except that objects are serialized as individual records, each prefixed by an ASCII Record Separator (<RS>, 0x1E), and each ending with an ASCII Line Feed character (\n, 0x0A). Note that each record can also contain any amount of newlines in its body, as long as it ends with a newline character before the next <RS> character.

In order to leverage the streaming capability, each qlog event is serialized and interpreted as an individual JSON Text Sequence record, that is appended as a new object to the back of an event stream or log file. Put differently, unlike default JSON, it does not require a document to be wrapped as a full object with "{ ... }" or "[...]".

This alternative record streaming approach cannot be accommodated by QlogFile (Figure 3). Instead, QlogFileSeq is defined in Figure 9, which notably includes only a single trace (TraceSeq) and omits an explicit "events" array. An example is provided in Figure 10. The "group_id" field can still be used on a per-event basis to include events from conceptually different sources in a single JSON-SEQ qlog file.

When mapping qlog to JSON-SEQ, the Media Type is "application/qlog+json-seq" per [RFC8091]. The file extension/suffix SHOULD be ".sqlog" (for "streaming" qlog).

While not specifically required by the JSON-SEQ specification, all qlog field names MUST be lowercase when serialized to JSON-SEQ.

In order to serialize all other CDDL-based qlog event and data structure definitions to JSON-SEQ, the official CDDL-to-JSON mapping defined in Appendix E of [CDDL] SHOULD be employed.

11.2.1. Supporting JSON Text Sequences in tooling

Note that JSON Text Sequences are not supported in most default programming environments (unlike normal JSON). However, several custom JSON-SEQ parsing libraries exist in most programming languages that can be used and the format is easy enough to parse with existing implementations (i.e., by splitting the file into its component records and feeding them to a normal JSON parser individually, as each record by itself is a valid JSON object).

11.3. JSON Interoperability

Some JSON implementations have issues with the full JSON format, especially those integrated within a JavaScript environment (e.g., Web browsers, NodeJS). I-JSON (Internet-JSON) is a subset of JSON for such environments; see [I-JSON]. One of the key limitations of JavaScript, and thus I-JSON, is that it cannot represent full 64-bit integers in standard operating mode (i.e., without using BigInt extensions), instead being limited to the range -(2^53)+1 to (2^53)-1.

To accommodate such constraints in CDDL, Appendix E of [CDDL] recommends defining new CDDL types for int64 and uint64 that limit their values to the restricted 64-bit integer range. However, some of the protocols that qlog is intended to support (e.g., QUIC, HTTP/3), can use the full range of uint64 values.

As such, to support situations where I-JSON is in use, seralizers MAY encode uint64 values using JSON strings. qlog parsers, therefore, SHOULD support parsing of uint64 values from JSON strings or JSON numbers unless there is out-of-band information indicating that neither the serializer nor parser are constrained by I-JSON.

11.4. Truncated values

For some use cases (e.g., limiting file size, privacy), it can be necessary not to log a full raw blob (using the hexstring type) but instead a truncated value. For example, one might only store 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 hexstring they specify, see for example Section 10. This not only ensures the original length can always be retrieved, 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 SHOULD 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. The main possible permutations are shown by example in Figure 41.

```
// both the content's value and its length are present
// (length is redundant)
{
    "content_length": 5,
    "content": "051428abff"
}

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

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

// both value and length are present, but the lengths
// do not match: the value was truncated to
// the first three bytes.
{
    "content_length": 5,
    "content": "051428"
}
```

Figure 41: Example for serializing truncated hexstrings

11.5. Optimization of serialized data

Both the JSON and JSON-SEQ 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 JSON(-SEQ) (de)serialization performance is typically (slightly) lower than that of more optimized and predictable formats. Both aspects present challenges to large scale setups, though they may still be practical to deploy; see [ANRW-2020]. JSON and JSON-SEQ compress very well using commonly-available algorithms such as GZIP or Brotli.

During the development of qlog, a multitude of alternative formatting and optimization options were assessed and the results are summarized on the qlog github repository (https://github.com/quiclog/internet-drafts/issues/30#issuecomment-617675097).

Formal definition of additional qlog formats or encodings that use the optimization techniques described here, or any other optimization technique is left to future activity that can apply the following guidelines.

In order to help tools correctly parse and process serialized qlog, it is RECOMMENDED that new formats also define suitable file extensions and media types. This provides a clear signal and avoids the need to provide out-of-band information or to rely on heuristic fallbacks; see Section 13.

12. 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.

12.1. Set file output destination via an environment variable

To provide users control over where and how qlog files are created, two environment variables are defined. 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 7.3) 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 42.

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 42: Environment variable examples for a QUIC implementation

13. Tooling requirements

Tools ingestion qlog MUST indicate which qlog version(s), qlog format(s), qlog file and event schema(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 11.1). Additionally, they SHOULD indicate exactly which values for and properties of the name (namespace:event_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 file and event schemas. 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.

14. Security and privacy considerations

Protocols such as TLS [RFC8446] and QUIC [RFC9000] offer secure protection for the wire image [RFC8546]. Logging can reveal aspects of the wire image that would ordinarily be protected, creating tension between observability, security and privacy, especially if data can be correlated across data sources.

qlog permits logging of a broad and detailed range of data. Operators and implementers are responsible for deciding what data is logged to address their requirements and constraints. As per [RFC6973], operators must be aware that data could be compromised, risking the privacy of all participants. Where entities expect protocol features to ensure data privacy, logging might unknowingly be subject to broader privacy risks, undermining their ability to assess or respond effectively.

14.1. Data at risk

qlog operators and implementers need to consider security and privacy risks when handling qlog data, including logging, storage, usage, and more. The considerations presented in this section may pose varying risks depending on the the data itself or its handling.

The following is a non-exhaustive list of example data types that could contain sensitive information that might allow identification or correlation of individual connections, endpoints, users or sessions across qlog or other data sources (e.g., captures of encrypted packets):

* IP addresses and transport protocol port numbers.

* Session, Connection, or User identifiers e.g., QUIC Connection IDs Section 9.5 of [RFC9000]).

* System-level information e.g., CPU, process, or thread identifiers.

* Stored State e.g., QUIC address validation and retry tokens, TLS session tickets, and HTTP cookies.

* TLS decryption keys, passwords, and HTTP-level API access or authorization tokens.

* High-resolution event timestamps or inter-event timings, event counts, packet sizes, and frame sizes.

* Full or partial raw packet and frame payloads that are encrypted.

* Full or partial raw packet and frame payloads that are plaintext e.g., HTTP Field values, HTTP response data, or TLS SNI field values.

14.2. Operational implications and recommendations

Operational considerations should focus on authorizing capture and access to logs. Logging of Internet protocols using qlog can be equivalent to the ability to store or read plaintext communications. Without a more detailed analysis, all of the security considerations of plaintext access apply.

It is recommended that qlog capture is subject to access control and auditing. These controls should support granular levels of information capture based on role and permissions (e.g., capture of more-sensitive data requires higher privileges).

It is recommended that access to stored qlogs is subject to access control and auditing.

End users might not understand the implications of qlog to security or privacy, and their environments might limit access control techniques. Implementations should make enabling qlog conspicuous (e.g., requiring clear and explicit actions to start a capture) and resistant to social engineering, automation, or drive-by attacks; for example, isolation or sandboxing of capture from other activities in the same process or component.

It is recommended that data retention policies are defined for the storage of qlog files.

It is recommended that qlog files are encrypted in transit and at rest.

14.3. Data minimization or anonymization

Applying data minimization or anonymization techniques to qlog might help address some security and privacy risks. However, removing or anonymizing data without sufficient care might not enhance privacy or security and could diminish the utility of qlog data.

Operators and implementers should balance the value of logged data with the potential risks of voluntary or involuntary disclosure to trusted or untrusted entities. Importantly, both the breadth and depth of the data needed to make it useful, as well as the definition of entities depend greatly on the intended use cases. For example, a research project might be tightly scoped, time bound, and require participants to explicitly opt in to having their data collected with the intention for this to be shared in a publication. Conversely, a server administrator might desire to collect telemetry, from users whom they have no relationship with, for continuing operational needs.

The most extreme form of minimization or anonymization is deleting a field, equivalent to not logging it. qlog implementations should offer fine-grained control for this on a per-use-case or per-connection basis.

Data can undergo anonymization, pseudonymization, permutation, truncation, re-encryption, or aggregation; see Appendix B of [DNS-PRIVACY] for techniques, especially regarding IP addresses. However, operators should be cautious because many anonymization methods have been shown to be insufficient to safeguard user privacy or identity, particularly with large or easily correlated data sets.

Operators should consider end user rights and preferences. Active user participation (as indicated by [RFC6973]) on a per-qlog basis is challenging but aligning qlog capture, storage, and removal with existing user preference and privacy controls is crucial. Operators should consider agressive approaches to deletion or aggregation.

The most sensitive data in qlog is typically contained in RawInfo type fields (see Section 10). Therefore, qlog users should exercise caution and limit the inclusion of such fields for all but the most stringent use cases.

15. IANA Considerations

IANA is requested to register a new entry in the "IETF URN Sub-namespace for Registered Protocol Parameter Identifiers" registry ([RFC3553])":

Registered Parameter Identifier: qlog

Reference: This Document

IANA Registry Reference: <https://www.iana.org/assignments/qlog>

IANA is requested to create the "qlog log file schema URIs" registry at https://www.iana.org/assignments/qlog for the purpose of registering log file schema. It has the following format/template:

Log File Schema URI: [the log file schema identifier]

Description: [a description of the log file schema]

Reference: [to a specification defining the log file schema]

This document furthermore adds the following two new entries to the "qlog log file schema URIs" registry:

Log File Schema URI	Description	Reference
urn:ietf:params:qlog:file:contained	Concrete log file schema that can contain several traces from multiple vantage points.	Section 4
urn:ietf:params:qlog:file:sequential	Concrete log file schema containing a single trace, optimized for seqential read and write access.	Section 5

Table 1

IANA is requested to create the "qlog event schema URIs" registry at https://www.iana.org/assignments/qlog for the purpose of registering event schema. It has the following format/template:

Event schema URI: [the event schema identifier]

Namespace: [the identifier of the namespace that this event schema either defines or extends]

Event Types: [a comma-separated list of concrete event types defined in the event schema]

Description: [a description of the event schema]

Reference: [to a specification defining the event schema definition]

This document furthermore adds the following two new entries to the "qlog event schema URIs" registry:

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

Namespace loglevel

Event Types error,warning,info,debug,verbose

Description: Well-known logging levels for free-form text.

Reference: Section 9.1

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

Namespace simulation

Event Types scenario,marker

Description: Events for simulation testing.

Reference: Section 9.2

16. References

16.1. Normative References

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

[DNS-PRIVACY]
Dickinson, S., Overeinder, B., van Rijswijk-Deij, R., and A. Mankin, "Recommendations for DNS Privacy Service Operators", BCP 232, RFC 8932, DOI 10.17487/RFC8932, October 2020, <https://www.rfc-editor.org/rfc/rfc8932>.

[I-JSON] Bray, T., Ed., "The I-JSON Message Format", RFC 7493, DOI 10.17487/RFC7493, March 2015, <https://www.rfc-editor.org/rfc/rfc7493>.

[JSON] 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/rfc/rfc8259>.

[JSON-Text-Sequences]
Williams, N., "JavaScript Object Notation (JSON) Text Sequences", RFC 7464, DOI 10.17487/RFC7464, February 2015, <https://www.rfc-editor.org/rfc/rfc7464>.

[RFC2046] Freed, N. and N. Borenstein, "Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types", RFC 2046, DOI 10.17487/RFC2046, November 1996, <https://www.rfc-editor.org/rfc/rfc2046>.

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

[RFC3339] Klyne, G. and C. Newman, "Date and Time on the Internet: Timestamps", RFC 3339, DOI 10.17487/RFC3339, July 2002, <https://www.rfc-editor.org/rfc/rfc3339>.

[RFC3553] Mealling, M., Masinter, L., Hardie, T., and G. Klyne, "An IETF URN Sub-namespace for Registered Protocol Parameters", BCP 73, RFC 3553, DOI 10.17487/RFC3553, June 2003, <https://www.rfc-editor.org/rfc/rfc3553>.

[RFC3986] Berners-Lee, T., Fielding, R., and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax", STD 66, RFC 3986, DOI 10.17487/RFC3986, January 2005, <https://www.rfc-editor.org/rfc/rfc3986>.

[RFC6839] Hansen, T. and A. Melnikov, "Additional Media Type Structured Syntax Suffixes", RFC 6839, DOI 10.17487/RFC6839, January 2013, <https://www.rfc-editor.org/rfc/rfc6839>.

[RFC6973] Cooper, A., Tschofenig, H., Aboba, B., Peterson, J., Morris, J., Hansen, M., and R. Smith, "Privacy Considerations for Internet Protocols", RFC 6973, DOI 10.17487/RFC6973, July 2013, <https://www.rfc-editor.org/rfc/rfc6973>.

[RFC7464] Williams, N., "JavaScript Object Notation (JSON) Text Sequences", RFC 7464, DOI 10.17487/RFC7464, February 2015, <https://www.rfc-editor.org/rfc/rfc7464>.

[RFC8091] Wilde, E., "A Media Type Structured Syntax Suffix for JSON Text Sequences", RFC 8091, DOI 10.17487/RFC8091, February 2017, <https://www.rfc-editor.org/rfc/rfc8091>.

[RFC8126] Cotton, M., Leiba, B., and T. Narten, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 8126, DOI 10.17487/RFC8126, June 2017, <https://www.rfc-editor.org/rfc/rfc8126>.

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

[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/rfc/rfc8259>.

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

16.2. Informative References

[ANRW-2020] Marx, R., Piraux, M., Quax, P., and W. Lamotte, "Debugging QUIC and HTTP/3 with qlog and qvis", September 2020, <https://qlog.edm.uhasselt.be/anrw/>.

[QLOG-H3] Marx, R., Niccolini, L., Seemann, M., and L. Pardue, "HTTP/3 qlog event definitions", Work in Progress, Internet-Draft, draft-ietf-quic-qlog-h3-events-11, 7 July 2025, <https://datatracker.ietf.org/doc/html/draft-ietf-quic-qlog-h3-events-11>.

[QLOG-QUIC] Marx, R., Niccolini, L., Seemann, M., and L. Pardue, "QUIC event definitions for qlog", Work in Progress, Internet-Draft, draft-ietf-quic-qlog-quic-events-11, 7 July 2025, <https://datatracker.ietf.org/doc/html/draft-ietf-quic-qlog-quic-events-11>.

[RFC8446] Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", RFC 8446, DOI 10.17487/RFC8446, August 2018, <https://www.rfc-editor.org/rfc/rfc8446>.

[RFC8546] Trammell, B. and M. Kuehlewind, "The Wire Image of a Network Protocol", RFC 8546, DOI 10.17487/RFC8546, April 2019, <https://www.rfc-editor.org/rfc/rfc8546>.

Acknowledgements

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

Thanks to Jana Iyengar, Brian Trammell, Dmitri Tikhonov, Stephen Petrides, Jari Arkko, Marcus Ihlar, Victor Vasiliev, Mirja Kuehlewind, Jeremy Laine, Kazu Yamamoto, Christian Huitema, Hugo Landau, Will Hawkins, Mathis Engelbart, Kazuho Oku, and Jonathan Lennox for their feedback and suggestions.

Change Log

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

Since draft-ietf-quic-qlog-main-schema-12:

* Changed Path and related fields to Tuple (#491)

* Replaced all lenght fields with raw.length (#495)

Since draft-ietf-quic-qlog-main-schema-10:

* Multiple editorial changes

* Remove protocol_types and move event_schemas to Trace and TraceSeq (#449)

Since draft-ietf-quic-qlog-main-schema-09:

* Renamed protocol_type to protocol_types (#427)

* Moved Trigger section. Purely editorial (#430)

* Removed the concept of categories and updated extension and event schema logic to match. Major change (#439)

* Reworked completely how we handle timestamps and clocks. Major change (#433)

Since draft-ietf-quic-qlog-main-schema-08:

* TODO (we forgot...)

Since draft-ietf-quic-qlog-main-schema-07:

* Added path and PathID (#336)

* Removed custom definition of uint64 type (#360, #388)

* ProtocolEventBody is now called ProtocolEventData (#352)

* Editorial changes (#364, #289, #353, #361, #362)

Since draft-ietf-quic-qlog-main-schema-06:

* Editorial reworking of the document (#331, #332)

* Updated IANA considerations section (#333)

Since draft-ietf-quic-qlog-main-schema-05:

* Updated qlog_version to 0.4 (due to breaking changes) (#314)

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

* Added ’system_info’ field (#305)

* Removed ’summary’ and ’configuration’ fields (#308)

* Editorial and formatting changes (#298, #303, #304, #316, #320, #321, #322, #326, #328)

Since draft-ietf-quic-qlog-main-schema-04:

* Updated RawInfo definition and guidance (#243)

Since draft-ietf-quic-qlog-main-schema-03:

* Added security and privacy considerations discussion (#252)

Since draft-ietf-quic-qlog-main-schema-02:

* No changes - new draft to prevent expiration

Since draft-ietf-quic-qlog-main-schema-01:

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

Since draft-ietf-quic-qlog-main-schema-00:

* Changed the streaming serialization format from NDJSON to JSON Text Sequences (#172)

* Added Media Type definitions for various qlog formats (#158)

* Changed to semantic versioning

Since draft-marx-qlog-main-schema-draft-02:

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

* Moved RawInfo, Importance, Generic events and Simulation events to this document.

* Added basic event definition guidelines

* Made protocol_type an array instead of a string (#146)

Since draft-marx-qlog-main-schema-01:

* 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)

* Removed "time_units" from Configuration. All times are now in ms instead (#95)

* Removed the "event_fields" setup for a more straightforward JSON format (#101,#89)

* Added a streaming option using the NDJSON format (#109,#2,#106)

* Described optional optimization options for implementers (#30)

* Added QLOGDIR and QLOGFILE environment variables, clarified the .well-known URL usage (#26,#33,#51)

* Overall tightened up the text and added more examples

Since draft-marx-qlog-main-schema-00:

* All field names are now lowercase (e.g., category instead of CATEGORY)

* Triggers are now properties on the "data" field value, instead of separate field types (#23)

* group_ids in common_fields is now just also group_id

Authors' Addresses

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

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

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

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