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

R. Marx
Hasselt University
2 November 2020

Main logging schema for qlog
draft-marx-qlog-main-schema-02

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

Copyright Notice

Table of Contents

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-HTTP3] for QUIC and HTTP/3-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 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.

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 (https://www.typescriptlang.org/).

This document describes how to employ JSON and NDJSON as textual serializations for qlog in Section 4. Other documents will describe how to utilize other concrete serialization options, though tips and requirements for these are also listed in this document (Section 4).

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

*   obj? : this object is optional

*   type1 | type2 : a union of these two types (object can be either type1 OR type2)

*   obj:type : this object has this concrete type

*   obj:array<type> : this object is an array of this type

*   class : defines a new type

*   // : single-line comment

The main data types are:

*   int8 : signed 8-bit integer

*   int16 : signed 16-bit integer

*   int32 : signed 32-bit integer

*   int64 : signed 64-bit integer

*   uint8 : unsigned 8-bit integer

*   uint16 : unsigned 16-bit integer

*   uint32 : unsigned 32-bit integer

*   uint64 : unsigned 64-bit integer

*   float : 32-bit floating point value

* double : 64-bit floating point value

* byte : an individual raw byte (8-bit) value (use array<byte> or the shorthand "bytes" to specify a binary blob)

* string : list of Unicode (typically UTF-8) encoded characters

* boolean : boolean

* enum: fixed list of values (Unless explicity defined, the value of an enum entry is the string version of its name (e.g., initial = "initial"))

* 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:

* Streamable, event-based logging

* Flexibility in the format, complexity in the tooling (e.g., few components are a MUST, tools need to deal with this)

* Extensible and pragmatic (e.g., no complex fixed schema with extension points)

* 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)

* Metadata is stored together with event data

3. The high level qlog schema

A qlog file should be able to contain several indivdual 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 "draft-02".

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 4) 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-02",
    "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 were 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 3.4.7.

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 (https://qvis.edm.uhasselt.be) 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 3.4.6), and "group_id" Section 3.4.7. As especially these later fields typically have identical values across individual event instances, they are normally logged separately in the "common_fields" (Section 3.4.8).

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 and HTTP/3 can be found in [QLOG-QUIC-HTTP3].

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-HTTP3].

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?: string,
    group_id?: string|uint32,

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

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

JSON serialization:

```
{
    time: 1553986553572,

    name: "transport:packet_sent",
    event: "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 occured. 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 succint 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":

* 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.

* 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.

* Relative: Specify a full "reference_time" timestamp (typically this is done up-front in "common_fields", see Section 3.4.8) 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 postprocesser 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 definitons for QUIC and HTTP/3, see [QLOG-QUIC-HTTP3].

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" field indicates to which protocol (or protocol "stack") 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_HTTP3" protocol_type value, see [QLOG-QUIC-HTTP3].

Typically however, all events in a single trace are of the same protocol, and this field is logged once in "common_fields", see Section 3.4.8.

3.4.5. custom fields

Note that qlog files can always contain custom fields (e.g., a per-event field indicating its privacy properties or path_id in multipath protocols) and assign custom values to existing fields (e.g., new categories/types for implemenation-specific events). Loggers are free to add such fields and field values and tools MUST either ignore these unknown fields or show them in a generic fashion.

3.4.6. 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 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 occuring. 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" | "unsupported_version"
}
```

Figure 12: Trigger example

3.4.7. 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_HTTPS2",
        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 3.4.8.

3.4.8. 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 uknown 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:

* time_format

* reference_time

* protocol_type

* 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. 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 4.3.

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 4.3).

4.1. 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.

4.1.1. 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 (https://en.wikipedia.org/wiki/Floating-point_arithmetic) 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 (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/BigInt) 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.

4.1.2. 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.

4.1.2.1. 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

4.1.3. 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 ([...])

Table 1

4.1.4. 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.

4.2. 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) (http://ndjson.org/).

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-02","title":"Name of this particular NDJSON qlog file (short)","description":"Description for this NDJSON qlog file (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.

4.2.1. 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 (http://ndjson.org/libraries.html) 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).

4.3. Other optimizated 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 (https://qlog.edm.uhasselt.be/anrw/)) 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 (https://github.com/quiclog/internet-drafts/issues/30#issuecomment-617675097). 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 4.3.2, 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 6), 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.

4.3.1. 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 (https://www.chromium.org/developers/design-documents/network-stack/netlog)) 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 4.3.2), these schemes are not inherently part of qlog.

4.3.2. 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 6).

4.3.3. 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 straighforward 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 (https://developers.google.com/protocol-buffers) (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 it 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 4.3.2). 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 compression the textual JSON equivalents directly (7%, see Section 4.3.2). As such, since compression is still needed to achieve optimal file size reductions event with binary formats, we feel the more flexible compressed textual JSON options are a better default for the qlog format in general.

4.3.4. 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 (https://github.com/quiclog/qlog).

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

format	qlog_format	extension
JSON Section 4.1	JSON	.qlog
NDJSON Section 4.2	NDJSON	.qlog
named headers Section 4.3.1	(ND)JSON.namedheaders	.qlog
dictionary Section 4.3.1	(ND)JSON.dictionary	.qlog
CBOR Section 4.3.3	(ND)JSON	.qlog.cbor
protobuf Section 4.3.3	protobuf	.qlog.protobuf
gzip Section 4.3.2	no change	.gz suffix
brotli Section 4.3.2	no change	.br suffix

Table 2

4.4. 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 (https://github.com/quiclog/qvis/tree/master/visualizations/src/components/filemanager/netlogconverter) converts chromium's internal dictionary-encoded JSON format to qlog. Finally, quictrace2qlog (https://github.com/quiclog/quictrace2qlog) 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 (https://qvis.edm.uhasselt.be) toolsuite supports a multitude of formats and qlog serializations.

5. 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.

5.1. 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 3.4.7) 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

5.2. 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 4.3.4). 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 7 before exposing qlogs to outside actors.

6. 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 4.1). 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 occurences 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.

7. 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 (https://huitema.wordpress.com/2020/07/21/scrubbing-quic-logs-for-privacy/)

8. IANA Considerations

TODO: primarily the .well-known URI

9. References

9.1. Normative References

[QLOG-QUIC-HTTP3]
Marx, R., Ed., "QUIC and HTTP/3 event definitions for qlog", Work in Progress, Internet-Draft, draft-marx-qlog-event-definitions-quic-h3-02, 2 November 2020, <https://tools.ietf.org/html/draft-marx-qlog-event-definitions-quic-h3-02>.

9.2. 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>.

Appendix A. Change Log

A.1. 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

A.2. 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

Appendix B. Design Variations

* Quic-trace (https://github.com/google/quic-trace) takes a slightly different approach based on protocolbuffers.

* Spindump (https://github.com/EricssonResearch/spindump) also defines a custom text-based format for in-network measurements

* Wireshark (https://www.wireshark.org/) 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

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.

Author’s Address

Robin Marx
Hasselt University

Email: robin.marx@uhasselt.be