QUIC R. Marx
Internet-Draft Hasselt University
Intended status: Standards Track July 03, 2019
Expires: January 4, 2020

Main logging schema for qlog
draft-marx-qlog-main-schema-00

Abstract

This document describes a high-level schema for a standardized endpoint 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.

Status of This Memo

This Internet-Draft will expire on January 4, 2020.

Copyright Notice

Table of Contents

1. Introduction

Feedback and discussion welcome at https://github.com/quiclog/internet-drafts

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].

2. Design Goals

The main tenets for the schema design are:

- Streamable, event-based
- Flexibility in the format, complexity in the tooling (e.g., few components are a MUST, tools need to deal with this)
- Extensible but 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)
- Explicit and human-readable

3. The High Level Schema

3.1. Top level container

To allow separate qlog traces to be contained within a single, encompassing qlog file, the top-level element in the qlog schema defines only a small set of fields and an array of component traces. Only the "qlog_version" and "traces" fields MUST be present. For this document, the "qlog_version" field MUST have a value of draft-00.

```
{
    "qlog_version": "draft-00",
    "title": "Name of this particular qlog file (short)",
    "description": "Description for this group of traces (long)",
    "summary": {
        ...
    }
    "traces": [...]
}
```

Figure 1: Top-level element

Typical logs will only contain a single element in the "traces" array. Multiple traces can then be combined into a single qlog file by taking the "traces" 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.

For example, for a test setup, we perform logging on the CLIENT, on the SERVER and on a single point on their common NETWORK path. Each

of these three logs is first created separately during the test. Afterwards, the three logs can be aggregated into a single qlog file.

3.2. Summary field

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, ...). 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:

```
"summary": {
    "trace_count":number, // amount of traces in this file
    "max_duration":string, // time duration of the longest trace
    "max_outgoing_loss_rate":number, // highest loss rate for outgoing packets
over all traces
    "total_event_count":number // total number of events across all traces
}
```

- TODO: are there any field semantics we should specify here?
- TODO: Will people actually use this? or will they store this info out-of-band (e.g., separate database for faster querying?)

3.3. Trace container

Each trace container encompasses a single conceptual trace. The exact 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, ...

In the normal use case, 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. However, a single trace could also combine events from a variety of vantage points or use cases (e.g., multiple QUIC connections or the same connection viewed from different points in the network).

The semantics and context of the trace can be deduced from the entries in the "common_fields" (specifically the "group_ids" field) and "event_fields" lists.

Only the "event_fields" and "events" fields MUST be present.

```
{
    "vantage_point": {
        "name": "backend-67",
        "type": "SERVER"
    },
    "title": "Name of this particular trace (short)",
    "description": "Description for this trace (long)",
    "configuration": {
        "time_offset": "offset in ms",
        "time_units": "ms" | "us"
    },
    "common_fields": (see below),
    "event_fields": (see below),
    "events": [...]
}
```

Figure 2: Trace container

3.3.1. vantage_point

This field describes the vantage point from which the trace originates. Its value is an object, with the following fields:

- name: an optional, user-chosen string (e.g., "NETWORK-1", "loadbalancer45", "reverseproxy@192.168.1.1", ...)

- type: one of three values: "SERVER", "CLIENT", "NETWORK".

  - CLIENT indicates an endpoint which initiates the connection.

  - SERVER indicates an endpoint which accepts the connection.

  - NETWORK indicates an observer in between CLIENT and SERVER.

- flow: one of two values: "CLIENT" or "SERVER".

  - This field is only required if type is "NETWORK".

  - CLIENT indicates that this vantage point follows client data flow semantics (a PACKET_TX goes in the direction of the SERVER).

  - SERVER indicates that this vantage point follow server data flow semantics (a PACKET_TX goes in the direction of the client).

The type field MUST be present. The flow field MUST be present if the type field has value "NETWORK". The name field is optional.

TODO (see issue 6): "NETWORK" should have a way to indicate what RX and TX mean (is current way enough? maybe identify endpoints by ID or 4-tuple etc.)

3.3.2. Title and Description

Both fields' values are generic strings, used for describing the contents of the trace. These can either be filled in automatically (e.g., showing the endpoint name and readable timestamp of the log), or can be filled manually when creating aggregated logs (e.g., qlog files that illustrate a specific problem across traces that want to include additional explanations for easier communication between teams, students, ...).

3.3.3. Configuration

We take into account that a log 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 log 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 standard, tool-independent configuration settings: "time_offset" and "time_units".

3.3.3.1. time_offset

time_offset indicates by how many units of time (see next section) 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.

3.3.3.2. time_units

Since timestamps can be stored in various granularities, this field allows to indicate whether storage happens in either milliseconds ("ms") or microseconds ("us"). If this field is not present, the default value is "ms". This configuration setting applies to all

other timestamps in the trace file as well, not just the "time_offset" field.

3.3.4. common_fields and event_fields

To reduce file size and make logging easier, the trace schema lists the names of the specific fields that are logged per-event up-front, instead of repeating the field name with each value, as is common in traditiona JSON. This is done in the "event_fields" list. This allows us to encode individual events as an array of values, instead of an object. To reduce file size even further, common event fields that have the same value for all events in this trace, are listed as name-value pairs in "common_fields".

For example, when logging events for a single QUIC connection, all events will share the same "original destination connection ID" (ODCID). This field and its value should be set in "common_fields", rather than "event_fields". However, if a single trace would contain events for multiple QUIC connections at the same time (e.g., a single, big output log for a server), the ODCID can be different across events, and SHOULD be part of "event_fields".

Examples comparing traditional JSON vs the qlog format can be found in Figure 3 and Figure 4. The events described in these examples are purely for illustration. Actual event type definitions for the QUIC and HTTP/3 protocols can be found in TODO.

```
   {
       "events": [{
               "group_id": "127ecc830d98f9d54a42c4f0842aa87e181a",
               "ODCID": "127ecc830d98f9d54a42c4f0842aa87e181a",
               "protocol_type": "QUIC_HTTP3",
               "time": 1553986553574,
               "CATEGORY": "TRANSPORT",
               "EVENT_TYPE": "PACKET_RX",
               "TRIGGER": "LINE",
               "DATA": [...]
           },{
               "group_id": "127ecc830d98f9d54a42c4f0842aa87e181a",
               "ODCID": "127ecc830d98f9d54a42c4f0842aa87e181a",
               "protocol_type": "QUIC_HTTP3",
               "time": 1553986553579,
               "CATEGORY": "APPLICATION",
               "EVENT_TYPE": "DATA_FRAME_NEW",
               "TRIGGER": "GET",
               "DATA": [...]
           },
           ...
       ]
   }
```

Figure 3: Traditional JSON

```
{
    "common_fields": {
        "group_id": "127ecc830d98f9d54a42c4f0842aa87e181a",
        "ODCID": "127ecc830d98f9d54a42c4f0842aa87e181a",
        "protocol_type":  "QUIC_HTTP3",
        "reference_time": "1553986553572"
    },
    "event_fields": [
        "relative_time",
        "CATEGORY",
        "EVENT_TYPE",
        "TRIGGER",
        "DATA"
    ],
    "events": [[
            2,
            "TRANSPORT",
            "PACKET_RX",
            "LINE",
            [...]
        ],[
            7,
            "APPLICATION",
            "DATA_FRAME_NEW",
            "GET",
            [...]
        ],
        ...
    ]
}
```

Figure 4: qlog optimized JSON

The main field names that can be included in these fields are defined in Section 3.4.

Given that qlog is intended to be a flexible format, unknown field names in both "common_fields" and "event_fields" MUST be disregarded by the user (i.e., the presence of an uknown field is explicitly NOT an error).

This approach makes line-per-line logging easier and faster, as each log statement only needs to include the data for the events, not the field names. Events can also be logged and processed separately, as part of a contiguous event-stream.

3.3.4.1. common_fields format

An object containing pairs of "field name"-"field value". Fields included in "common_fields" indicate that these field values are the same for each event in the "events" array (with the exception of the "group_ids" field, see Section 3.4.2)

If even one event in the trace does not adhere to this convention, that field name should be in "event_fields" instead, and the value logged per event. An alternative route is to include the most commonly seen value in "common_fields" and then include the deviating field value in the generic "data" field for each non-confirming event. However, these semantics are not defined in this document.

3.3.4.2. event_fields format

An array of field names (plain strings). Field names included in "event_fields" indicate that these field names are present *in this exact order* for each event in the "events" array. Each individual event then only has to log the corresponding values for those fields in the correct order.

3.4. Field name semantics

This section lists pre-defined, reserved field names with specific semantics and expected corresponding value formats.

Only a time-based field (see Section 3.4.1), the EVENT_TYPE field and the DATA field are mandatory. Typical setups will log reference_time, protocol_type and group_id in "common_fields" and relative_time, CATEGORY, EVENT_TYPE, TRIGGER and DATA in "event_fields".

Other field names are allowed, both in "common_fields" and "event_fields", but their semantics depend on the context of the log usage (e.g., for QUIC, the ODCID field is used).

3.4.1. time, delta_time and reference_time + relative_time

There are three main modes for logging time:

- Include the full timestamp with each event ("time"). This approach uses the largest amount of characters.

- Delta-encode each time value on the previously logged value ("delta_time"). The first event can log the full timestamp. This approach uses the least amount of characters.

- Specify a full "reference_time" timestamp up-front in "common_fields" and include only relatively-encoded values based on this reference_time with each event ("relative_time"). 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.

```
The time approach will use:
1500, 1505, 1522, 1588

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

The relative_time approach will:
- set the reference_time to 1500 in "common_fields"
- use: 0, 5, 22, 88
```

Figure 5: Three different approaches for logging timestamps

3.4.2. group_id and group_ids

A single Trace can contain events from a variety of sources, belonging to for example a number of individual QUIC connections. For tooling considerations, it is necessary to have a well-defined way to split up events belonging to different logical groups into subgroups for visualization and processing. For example, if one type of log uses 4-tuples as identifiers and uses a field name "four_tuple" and another uses "ODCID", there is no way to know for generic tools which of these fields should be used to create subgroups. As such, qlog uses the generic "group_id" field to circumvent this issue.

The "group_id" field can be any type of valid JSON object, but is typically a string or integer. For more complex use cases, the group_id could become a complex object with several fields (e.g., a 4-tuple). In those cases, it would be wasteful to log these values in full every single time. This would also complicate tool-based processing. As a solution, qlog allows the extraction of group_id values into a separate "group_ids" field in the "common_fields", consisting of an array of the various present group ids for this trace. If this field is present, per-event "group_id" values are regarded as indices into the "group_ids" array. This is useful if the group_ids are known up-front or the qlog trace can be generated from a more verbose format afterwards. If this is not the case, it is acceptable to just log the complex objects as the "group_id" for

each event. Both use cases are demonstrated in Figure 6 and Figure 7.

Since "group_id" and "group_ids" are generic names, they convey little of the semantics to the casual reader. It is best practice to also include a per use case additional field to the "common_fields" with a semantic name, that has the same value as the "group_id" or "group_ids" field. For example, see the "ODCID" field in Figure 4 and the "four_tuples" field in Figure 7.

TODO: maybe just make group_ids or group_id reference the named field instead? e.g., "group_id": "ODCID"

TODO: for the simple use case (e.g., just 1 QUIC connection in the trace), MUST a trace include a group_id? maybe yes: the ODCID? (ODCID because the normal connection IDs can change during the QUIC connection).

```
{
    "common_fields": {
        "protocol_type":  "QUIC_HTTP3",
    },
    "event_fields": [
        "time",
        "group_id",
        "CATEGORY",
        "EVENT_TYPE",
        "TRIGGER",
        "DATA"
    ],
    "events": [[
            1553986553579,
            { "ip1": "2001:67c:1232:144:9498:6df6:f450:110b", "ip2": "2001:67c:
2b0:1c1::198", "port1": 59105, "port2": 80 }
            "TRANSPORT",
            "PACKET_RX",
            "LINE",
            [...]
        ],[
            1553986553588,
            { "ip1": "10.0.6.137", "ip2": "52.58.13.57", "port1": 56522,
"port2": 443 }
            "APPLICATION",
            "DATA_FRAME_NEW",
            "GET",
            [...]
        ],[
            1553986553598,
            { "ip1": "2001:67c:1232:144:9498:6df6:f450:110b", "ip2": "2001:67c:
2b0:1c1::198", "port1": 59105, "port2": 80 }
            "TRANSPORT",
            "PACKET_TX",
            "STREAM",
            [...]
        ],
        ...
    ]
}
```

Figure 6: Repeated complex group id

```
{
    "common_fields": {
        "protocol_type":  "QUIC_HTTP3",
        "group_ids": [
            { "ip1": "2001:67c:1232:144:9498:6df6:f450:110b", "ip2": "2001:67c:
2b0:1c1::198", "port1": 59105, "port2": 80 },
            { "ip1": "10.0.6.137", "ip2": "52.58.13.57", "port1": 56522,
"port2": 443 }
        ],
        "four_tuples": [
            { "ip1": "2001:67c:1232:144:9498:6df6:f450:110b", "ip2": "2001:67c:
2b0:1c1::198", "port1": 59105, "port2": 80 },
            { "ip1": "10.0.6.137", "ip2": "52.58.13.57", "port1": 56522,
"port2": 443 }
        ]
    },
    "event_fields": [
        "time",
        "group_id",
        "CATEGORY",
        "EVENT_TYPE",
        "TRIGGER",
        "DATA"
    ],
    "events": [[
            1553986553579,
            0
            "TRANSPORT",
            "PACKET_RX",
            "LINE",
            [...]
        ],[
            1553986553588,
            1
            "APPLICATION",
            "DATA_FRAME_NEW",
            "GET",
            [...]
        ],[
            1553986553598,
            0
            "TRANSPORT",
            "PACKET_TX",
            "STREAM",
            [...]
        ],
        ...
    ]
}
```

Figure 7: Indexed complex group id

3.4.3. CATEGORY and EVENT_TYPE

Both CATEGORY and EVENT_TYPE are separate, generic strings. CATEGORY allows a higher-level grouping of events per EVENT_TYPE.

For example, instead of having an EVENT_TYPE of value "QUIC_PACKET_TX", we instead have a CATEGORY of "QUIC" and EVENT_TYPE of "PACKET_TX". This allows for fast and high-level filtering based on CATEGORY and re-use of EVENT_TYPEs across categories.

3.4.4. TRIGGER

The TRIGGER field is a generic string. It indicates which type of event triggered this event to occur (alternately: which other event is the reason this event occured).

This additional information is needed in the case where a single EVENT_TYPE 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 logs messages might be wide and far between in time. The trigger field allows adding an additional hint as to the cause, even if the surrounding messages do not provide this context.

TODO: is this field needed at this level? see issue 7

3.4.5. DATA

The DATA field is a generic object (list of name-value pairs). It contains the per-event metadata and its form and semantics are defined per specific sort of event (typically per EVENT_TYPE, but possibly also by combination of CATEGORY, EVENT_TYPE and TRIGGER).

3.4.6. Event field values

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 draft-marx-qlog-event-definitions-quic-h3-00.

4. Tooling requirements

Tools MUST indicate which qlog version(s) they support. Additionally, they SHOULD indicate exactly which values for the CATEGORY, EVENT_TYPE and TRIGGER 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 errors for any field names and values in the qlog format that they do not recognize. Tools CAN indicate unknown event occurences within their context (e.g., marking unknown events on a timeline for manual interpretation by the logger).

5. Methods of Access

TBD : propose to use a .well-known URL to fetch logs from an endpoint / to send logs to.

6. Notes on Practical Use

TBD : discuss that implementations do not have to output qlog directly. It is good practice to log in whatever way you want, and then just write a transformer to qlog for use in tooling.

7. Security Considerations

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

8. IANA Considerations

TBD

9. References

9.1. Normative References

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

9.2. URIs

[1] https://github.com/google/quic-trace

[2] https://github.com/EricssonResearch/spindump

[3] https://www.wireshark.org/

Appendix A. Change Log

A.1. Since draft-marx-qlog-main-schema-00:

- None yet.

Appendix B. Design Variations

- Quic-trace [1] takes a slightly different approach based on protocolbuffers.
- Spindump [2] also defines a custom text-based format for in-network measurements
- Wireshark [3] 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 and Lucas Pardue for their feedback and suggestions.

Author's Address

Robin Marx
Hasselt University

Email: robin.marx@uhasselt.be