Debugging QUIC and HTTP/3 with qlog and qvis

Dominik von Künßberg, Benedikt Jaeger*
*Chair of Network Architectures and Services, Department of Informatics
Technical University of Munich, Germany
Email: dominikvon.kuenssberg@tum.de, jaeger@net.in.tum.de

***Abstract*—The powerful properties of the QUIC and HTTP/3 protocols make debugging and inspecting them a challenging task. The qlog format and the qvis toolsuite have been introduced to facilitate this problem. We give an overview of both the format and the visualization tool, introducing and assessing their respective capabilities.**

***Index Terms*—software-defined networks, measurement, high-speed networks**

1. Introduction

The QUIC protocol is a transport protocol designed to offer lower latency for HTTP traffic while also meeting security requirements by encrypting its packets, as described by Langley et al. [1]. Built on UDP, it forms the basis for HTTP/3. Lower latency is attributed to several things: firstly, it streamlines the amount of handshakes needed to establish a secure connection by exchanging cryptographic keys and certificates directly in the initial handshake [1]. It also identifies connections by a connection ID instead of the IP / port 5-tuple, allowing for immediate reconnection to a server after changing IP addresses [1]. To avoid head-of-line-problems like present in the TCP+TLS+HTTP/2 stack, QUIC allows multiple bidirectional streams within a QUIC connection which are independent of each other [1]. Lastly, QUIC packets are entirely encrypted except for fields necessary for routing, forwarding and decrypting the packet [1].

While this is an effective way to protect the packet's data, it makes the protocol difficult to analyze and debug. As there are various implementations of QUIC and its standardization is still ongoing [2], analysis and debugging are indispensable tools to verify the protocol's behavior and find bugs.

Capturing the available metadata from packets in transit alone is not sufficient because fields containing states necessary for analysis are encrypted [3]. Temporarily de- and encrypting the packets while in transit to extract the necessary log information is out of the question as this exposes the full payload and requires the respective session keys [3]. The only places where the later encrypted states are available are the endpoints which send and receive the packets [3]. Logging mechanisms have been implemented, however each is specific to their own implementation of the protocol which makes them difficult to parse [3] for further use.

To alleviate this problem, Marx et al. propose a logging format called qlog [3]. Qlog is based on JSON [3] which allows it to be used across implementations, independent of language-specific characteristics. Each qlog event is characterized by a timestamp, a category, the event type and type-specific data [3]. This format makes it easily extensible: to log a specific type of event which is not yet present, it can simply be added. Qlog files from different connection endpoints can also be aggregated into one single qlog file [3].

Logging events to analyze the performance and behaviour of the QUIC protocol is certainly helpful, however it might be hard to extract the needed information from textual logs only. Because of this, Marx et al. also created the tool qvis to visualize qlogs [3]. This is especially helpful combined with qlog's ability to combine logs from different endpoints; qvis is then able to visualize relations between the endpoints accurately such as packet loss, packet order etc [3].

This paper aims to outline the most important aspects of the qlog format and the qvis visualization tool. We present the qlog format in Section 2 and address how data is collected and its scalability in Section 3. In Section 4, we introduce the qvis toolsuite and its scalability. Section 5 assesses how the qlog format compares to the pcap format commonly used in the TCP+TLS+HTTP/2 stack. In Section 6, we conclude that both qlog and qvis are powerful tools for debugging the QUIC protocol and summarize future plans for qvis.

2. Qlog

The qlog format has so far been defined in two IETF drafts, one describing the general high-level format of qlog [4] and the other defining events specific to QUIC and HTTP/3 [5]. As qlog is a flexible and general format, it can also be used for protocols other than QUIC such as DNS or the TCP+TLS stack by defining the events in the implementation accordingly [3].

Fields inside a qlog file follow a JSON-like format. The basic format is an `object:type` pair. Available standard types are signed and unsigned integers with lengths varying from 8 to 64 bits, floats and doubles, strings, bytes (raw 8 bit long values), booleans, enums and any, which can represent any data type. Additional notations are listed in the Internet Draft for qlog [4].

Every qlog file consists of one top-level file which must contain a `qlog_version` field and an array containing traces [4, Section 3]. Further optional fields can be given such as `title`, `summary`, `description` and `qlog_format` [4, Section 3]. The summary can be useful to get a quick overview of aggregated information about

doi: 10.2313/NET-2021-05-1_06

all traces that have been logged, being able to list customizable features such as total lost packets, total number of events and whatever information may be needed in a specific use-case [4, Section 3.1].

2.1. Traces

A trace is a structure which contains the recorded events and additional metadata, however, it usually represents the data flow at a single endpoint [4, Section 3.3]. It must contain a `vantage_point` field to identify which type of endpoint it logged, and an array of events representing all logged events at this endpoint [4, Section 3.3]. Other optional fields allowing for more context are `title`, `description`, and, most importantly, the `common_fields` list [4, Section 3.3], which will be discussed in Section 2.2.

2.2. Events

Each event must at least contain the fields `timestamp`, `name`, and `data` [4, Section 3.4]. Usually it is useful to organize events by assigning them a `group_id`, a `protocol_type` and perhaps a `category` [4, Section 3.4]. Consequentially, fields such as these typically tend to stay the same for the majority of events from the same trace, and thus would need to be constantly logged anew [4, Section 3.4.8]. To avoid unnecessary duplicate data, a trace can contain the `common_fields` list, containing information which is shared by all events of that trace [4, Section 3.4.8]. The mentioned fields can then be omitted in the event itself.

Events can also contain so-called "triggers" in the `data` field [4, Section 3.4.6]. Triggers are a set of possible string values which indicate why an event has occurred [4, Section 3.4.6]. If the event occurs, the applicable trigger string is then included in the log. This gives a direct context to the occurrence of the event and eliminates the need of analyzing logs within roughly the same timeframe to find the reason [4, Section 3.4.6].

The QUIC specific events described in [5] have been divided into three categories: Core, Base, and Extra [5, Section 2.1].

Core events should be present in all qlog files and are used to log very basic information [5, Section 2.1]. Examples of Core events are `packet_sent`, `packet_received`, `version_information` and `packet_lost` [5, Sections 5.3, 5.4.5]. The `version_information` event logs the QUIC versions available for both client and server, as well as the version which has been selected [5, Section 5.3.1].

Base events can depend on Core events but are logged separately for the sake of clarity [5, Section 2.1]. They provide more detailed information which is relevant for debugging. Such events are for example `connection_started`, `packet_dropped`, `packet_buffered`, and `congestion_state_updated` [5, Sections 5.1.2, 5.3, 5.4.3].

Extra events are usually employed to observe the internal behavior of the protocol's implementation, rather than the protocol itself [5, Section 2.1]. Examples for Extra events include `server_listening`, `packets_acked`, `datagrams_sent` and `datagrams_received` [5, Sections 5.1.1, 5.3]. "Datagrams" in this case refers to UDP-datagrams [5, Section 5.3.10].

3. Qlog data collection

How and at which points qlog logs its data is entirely up to the implementation. Any necessary data structures need to be created as well as functions for forwarding and writing information to a qlog file. Coupled with the flexible format of qlogs, it allows for precise logs exactly where it is needed. As an example, the logging of a qlog event in the Go implementation is structured as follows. The file event.go contains and defines all possible events that can be logged [6]. Each event contains the needed and optional attributes which can be set [6]. A struct called `connectionTracer` acts as the trace explained in Section 2.1 [6]. It makes use of Go Channels to record events concurrent to program execution [6]. To avoid race conditions, a mutex is used on the `events` channel so that only one event can be recorded at a time [6]. For instance, when the server sends a version negotiation packet to the client, the `sentPacket` function of the `connectionTracer` is invoked, which in turn records the event and adds it to the `events` channel [6]. This behavior is essentially the same across all functions; when a function is called which necessitates logging, the respective function in the `connectionTracer` is called and adds the event to the log [6]. Upon stopping the server, the aggregated events are written to the qlog [6].

As this means that qlogs are held in memory and only written to the disk when the connection is terminated, this approach might cause unwanted occupation of memory when logging a large volume of events. As an example, the large demonstration file on the qvis website [7] representing a 100 MB download is 31 MB in size, while the qlog file for a 500 MB download mentioned in [3] is 276 MB. Assuming this can be scaled roughly linearly, logging a 10 GB download will then result in a qlog which is somewhere between 3,1 and 5,5 GB in memory before the connection is terminated. It is therefore important to keep this memory occupation in mind and evaluate which events actually need to be logged to minimize the resulting log size, especially when downloading and logging large quantities of data.

3.1. Scalability

In [3], Facebook employed qlog at internet scale and concluded that it "is two to three times as large" and "takes 50 % longer to serialize than their previous in-house binary format." In Facebook's case, this processing surplus was acceptable given the flexibility provided by qlog [3].

To compress qlog's size requirements while preserving the format's desirable properties, an optimized mode [3] was introduced. It relies on two aspects: reducing the initial size of qlogs and encoding the smaller qlogs more efficiently [3]. The former is accomplished by collecting repeated values in a dynamic dictionary [3]. The latter is achieved by using the CBOR (Concise Binary Object Representation) format to encode the qlogs and the generated dictionary [3]. CBOR is a binary format which preserves

JSON's key-value pairs in a concise manner and allows for faster processing than JSON [3].

The combination of these two methods results in significantly smaller file sizes. The qlog for a 500 MB download is usually 276 MB; utilizing the optimized mode results in a file about a third as large as the original one, ending up at 91 MB [3].

4. Qvis

Qvis encompasses a set of tools which visualizes qlog files and their data in an understandable and descriptive manner. It can handle qlog files which contain traces from several endpoints to deduce and display information from the provided data, such as round trip time, congestion control and more [7]. Qvis offers four visualization methods: sequence, congestion, multiplexing and packetization [7]. It also lists general statistics about the provided qlogs such as the number and types of events and frames [7]. Qvis is implemented mainly in TypeScript and Vue and intended to be used in a browser [7]. Scalability issues arising from this are discussed in Section 4.5.

4.1. Sequence Tool

The sequence tool generates a sequence diagram as shown in Figure 1a. The green squares on both sides represent events. If the event is neither `packet_received` nor `packet_sent`, the event name is added next to it [7]. Besides displaying the information contained in transmitted packets and their respective timestamps, all the green boxes, event names and packet information can be clicked which brings up the corresponding qlog file in plaintext, allowing for further, more detailed packet inspection [7].

4.2. Congestion Tool

The congestion tool shows two diagrams: one which shows the amount of data sent over time in bytes, and one displaying the round trip time [7]. What first appears as a slightly jagged line in the first diagram becomes clearer when zooming in; it shows the bursts of data being sent in blue and the acknowledgement of that data in green [7], as shown in Figure 1b. The gap between the blue and green blocks on the same height on the y-axis constitutes the round trip time [7]. The congestion tool, therefore, makes it easier to identify when data is being sent at a different rate indicated by a change of the slope of the graph [7]. It can also display crucial information such as the congestion window size and lost data [7].

4.3. Multiplexing Tool

The multiplexing tool shows how the data sent was divided among the existing QUIC streams (shown in Figure 1d) [7]. It assigns a color to each stream and displays the sent data as colored blocks strung along the timeline, each colored block indicating that the corresponding stream has been used to transmit data [7]. It also indicates which frames had to be resent underneath the corresponding parts of the diagram [7]. This makes it simple to identify unwanted behavior in the applied multiplexing strategy [7]. It is also possible to zoom into the string of blocks and hover over them to display the exact timestamp, utilized stream, number and packet size of the block that is being pointed at [7]. This is especially helpful when inspecting large qlogs.

Additionally, it is possible to enable two more supplementary diagrams: the waterfall and byterange diagram (waterfall diagram shown in Figure 1c) [7]. The waterfall diagram displays a colored bar for each stream between the first and last time it received a frame [7]. This makes it easier to determine roughly when a stream was active, especially when a large number of frames was transmitted [7]. The byterange diagram displays the range of bytes transmitted by the frames which are shown at the current zoom level [7].

4.4. Packetization Tool

The packetization tool visualizes how QUIC packets are composed of QUIC frames and HTTP/3 frames (shown in Figure 1e) [7]. Each layer represents one structure: in ascending order, those are QUIC packets, QUIC frames, HTTP/3 frames and the stream IDs present in the corresponding packet [7]. Headers within packets and frames are represented by taking up half the height of the line compared to the payload. Packet / Frame boundaries can be discerned by the alternating colors within each layer [7]. As with the other tools, it is possible to zoom in on a specific spot and hover over it to view packet or frame information [7].

4.5. Scalability

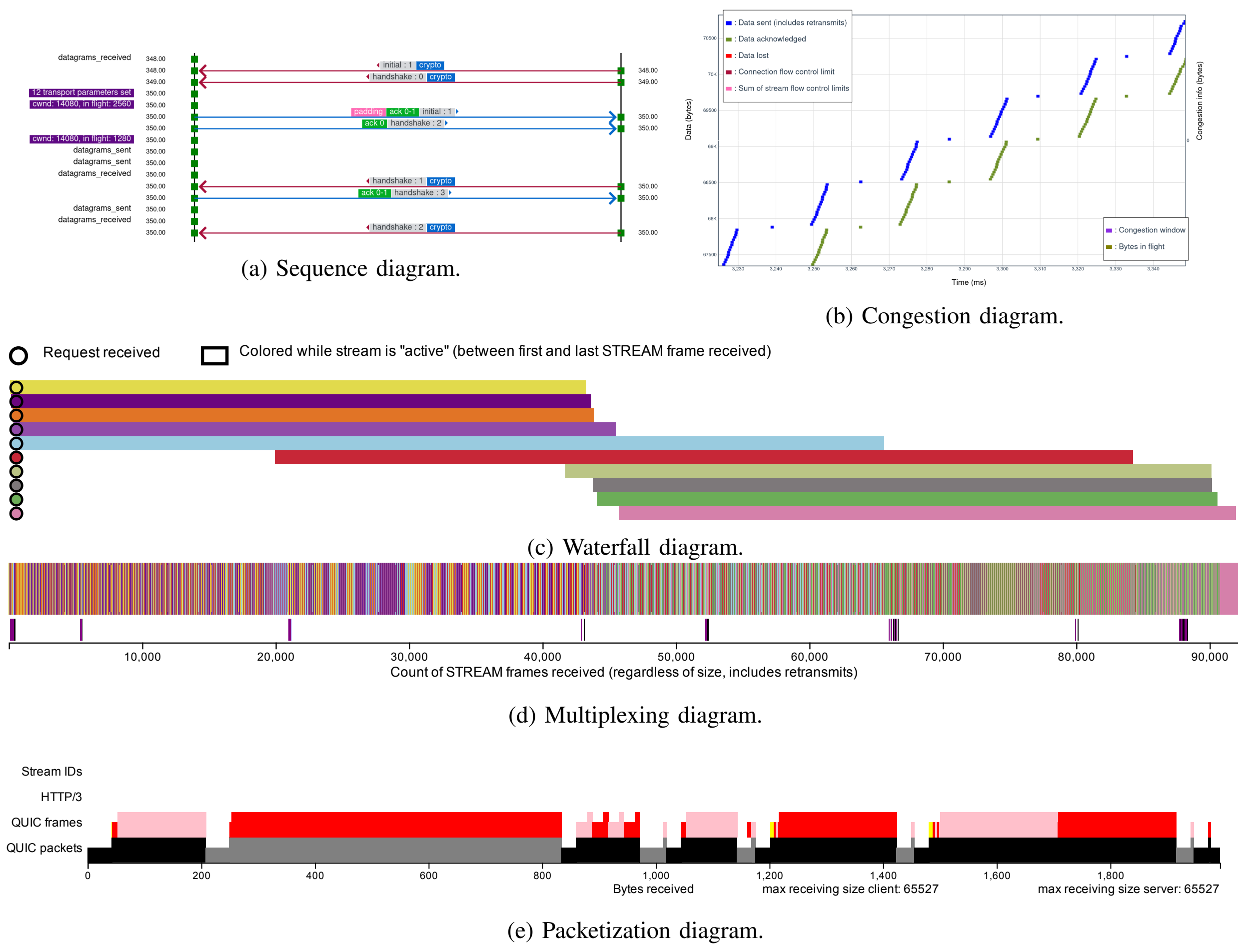
While it is possible to load large files in qvis and the authors of qvis describe in [3] that qvis "scales to loading hundreds of MB in JSON", it significantly impacts the performance of the tool. It is recommended to use a Chromium-based browser, as using another might affect performance even more [8].

Loading the demonstration file of 31 MB representing a 100 MB download [7] is certainly possible but switching between the different tools, using the zoom function to view packet details and other actions noticeably slow down the web browser. Tools especially affected are the sequence, multiplexing and packetization tools.

We observed that the sequence tool initially takes between 10 and 15 seconds to load the entries. However, once everything is loaded, the tools work perfectly fine.

The packetization tool also takes about the same time to initially load as the sequence tool. The congestion tool is the quickest to respond of all tools, the zoom works without delay. This is due to the fact that it uses canvas-based rendering [8].

Both the multiplexing and packetization tools share a performance issue with large files concerning the zoom function. Zooming in becomes more important as qlogs get bigger to analyze sections of the graph more closely. To dissect this issue, it is helpful to analyze how the depiction of the diagrams is implemented. Both tools use rendering of scalable vector graphics (SVG) to display the diagrams [8]. Each packet / frame is a separate SVG entity [8]. When zooming in or out, the dimensions of every entity has to be recalculated, which is slow with



(a) Sequence diagram.

(b) Congestion diagram.

(c) Waterfall diagram.

(d) Multiplexing diagram.

(e) Packetization diagram.

Figure 1: Qvis diagrams.

such big qlogs as this results in tens of millions of SVG entities being resized [8]. Therefore, this large workload is especially noticeable when using the packetization tool, as it contains far more SVG entities than the multiplexing diagram [7].

The SVG rendering approach was chosen because hover effects to display packet / frame information is easier to implement this way compared to the canvas-based rendering used by the congestion tool [8]. However, there are plans to port the remaining tools to the same rendering method for performance reasons [8].

5. Comparison with TCP+TLS+HTTP/2

The TCP+TLS+HTTP/2 (TTH) stack is most commonly debugged with tools such as Wireshark [9] or tcptrace [10]. For this purpose, the log consists of timestamps and the captured packets exactly as they were represented during transmission [3]. As the TTH stack shows most information necessary for debugging in the (unencrypted) headers of the packets, this is sufficient. For QUIC, this would not work as important metadata for debugging purposes such as frame numbers, frame type and stream IDs are in the encrypted section of the packet [3]. This makes a direct analysis of packets similar to that of the TTH stack regarding these properties infeasible.

In terms of log size, pcap files created with e.g. Wireshark can be of varying size depending on the applied options. With default settings, the pcap file for a 500 MB download will exceed 500 MB, as all packets are directly ingested into the log file. However, there are options to limit the capture size of each incoming packet [11], dropping most of the payload. This can dramatically decrease the file size. Measurements showed that when downloading a 500 MB file using Wireshark with default settings results in a pcap file of 550 MB. Restricting the size of each logged packet to 100 B to account for headers, the pcap file size drops to 65 MB.

The qlog file for a 500 MB download is 276 MB or 91 MB [3] when using the optimized mode explained in Section 3.1. This is evidently a noticeable difference in size.

Despite this, qlog has the advantage of offering a more detailed analysis of internal variables such as congestion window, lost packets and bytes in flight [3] in comparison to TCP traces and being able to visualize them accordingly with qvis via the congestion tool [3].

6. Conclusion

Qlog is a powerful logging tool which has tremendous potential for debugging internet protocols, QUIC in particular. Its ability to define custom events and to combine multiple traces into one qlog, paired with the terrific visualization capabilities of qvis, makes it a solid basis for anyone debugging QUIC.

Porting qvis to native code for better performance is currently not planned by the original developers of the tool [8]. While one of the goals is to write qlog importers for existing native tools such as Windows Performance Analyzer, this is not planned for the immediate future [8]. However, the performance issues due to SVG rendering are being worked on as it is planned to convert the respective tools to canvas-based rendering [8].

References

[1] A. Langley, A. Riddoch, A. Wilk, A. Vicente, C. B. Krasic, C. Shi, D. Zhang, F. Yang, F. Kouranov, I. Swett, J. Iyengar, J. Bailey, J. C. Dorfman, J. Roskind, J. Kulik, P. G. Westin, R. Tenneti, R. Shade, R. Hamilton, V. Vasiliev, and W.-T. Chang, "The QUIC transport protocol: Design and internet-scale deployment," 2017.

[2] D. Madariaga, L. Torrealba, J. Madariaga, J. Bermúdez, and J. Bustos-Jiménez, "Analyzing the adoption of QUIC from a mobile development perspective," in *Proceedings of the Workshop on the Evolution, Performance, and Interoperability of QUIC*, ser. EPIQ '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 35–41. [Online]. Available: https://doi-org/10.1145/3405796.3405830

[3] R. Marx, M. Piraux, P. Quax, and W. Lamotte, "Debugging QUIC and HTTP/3 with qlog and qvis," in *Proceedings of the Applied Networking Research Workshop*, ser. ANRW '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 58–66. [Online]. Available: https://doi.org/10.1145/3404868.3406663

[4] R. Marx, "Main logging schema for qlog," Internet Engineering Task Force, Internet-Draft, 2020, work in Progress. [Online]. Available: https://quiclog.github.io/internet-drafts/draft-marx-qlog-main-schema.html

[5] ——, "Quic and http/3 event definitions for qlog," Internet Engineering Task Force, Internet-Draft, 2020, work in Progress. [Online]. Available: https://quiclog.github.io/internet-drafts/draft-marx-qlog-event-definitions-quic-h3.html

[6] L. Clemente, "A QUIC implementation in pure go," 2020. [Online]. Available: https://github.com/lucas-clemente/quic-go

[7] R. Marx, "qvis: tools and visualizations for QUIC and HTTP/3," 2020, https://qvis.edm.uhasselt.be/.

[8] ——, "Qvis performance," 2020. [Online]. Available: https://github.com/quiclog/qvis/issues/38

[9] "Wireshark," 2020. [Online]. Available: https://www.wireshark.org/

[10] "Tcptrace," 2020. [Online]. Available: https://linux.die.net/man/1/tcptrace

[11] "Wireshark documentation," 2020. [Online]. Available: https://www.wireshark.org/docs/wsug_html/