Debugging Modern Web Protocols with qlog

Robin Marx  
robin.marx@uhasselt.be  
Hasselt University – tUL – EDM  
Diepenbeek - Belgium

Maxime Piraux  
maxime.piraux@uclouvain.be  
UCLouvain  
Belgium

Peter Quax  
peter.quax@uhasselt.be  
Hasselt University – tUL – Flanders  
Make – EDM  
Diepenbeek - Belgium

ABSTRACT

The QUIC and HTTP/3 protocols are powerful but complex and difficult to debug and analyse. Our previous work proposed the qlog format for structured endpoint logging to aid in taming this complexity. This follow-up study evaluates the real-world implementations, uses and deployments of qlog and our associated qvis tooling in academia and industry. Our survey among 28 QUIC experts shows high community involvement, while Facebook confirms qlog can handle Internet scale. Lessons learned from researching 16 QUIC+HTTP/3 and five TCP+TLS+HTTP/2 implementations demonstrate that qlog and qvis are essential tools for performing root-cause analysis when debugging modern Web protocols.

1 INTRODUCTION & MOTIVATION

The new QUIC and HTTP/3 (H3) protocols [16, 46] provide a large number of exciting features (such as 0-RTT handshakes and connection migration), but with great power comes great complexity. Spread over 440 pages in ten documents [90], the protocols are challenging to understand and implement. In 2018, by working on our own implementations [29, 60, 72], it became evident to us that the QUIC community would need extensive tooling and visualizations to help debug and validate their systems. These tools would ideally be re-usable across codebases, in turn requiring a common input data format. Looking at tools for established protocols like the TCP+TLS+HTTP/2 stack (e.g., wireshark, tcptrace, and captcp [10, 58, 61]), we see they ingest the protocols’ wire image directly (e.g., via (decrypted) packet capture (pcap) files). While this can work for QUIC, it is suboptimal during initial implementation, as the wire image lacks large pieces of internal state crucial to debugging complex components (e.g., congestion control variables). This data was at that time available only in ad-hoc command line logs, which are difficult to interpret, highly implementation-specific, and challenging to parse.

To combat this issue, we proposed qlog, a structured endpoint logging format [58, 61]. It standardizes the logging of QUIC and H3 events and internal state (e.g., why a packet was marked lost, how data moves between layers) in a machine-readable, JSON-based format. We also implemented several qlog-compatible tools in our qvis toolsuite [55], to trigger implementer interest. Yet at that time, it was firstly difficult to estimate how essential these tools would really be for QUIC debugging. Secondly and subsequently, it was unclear if QUIC developers were willing to adopt qlog. These are two of the three questions this follow-up work aims to answer.

The third question is whether qlog can scale, and be used not only to debug the initial implementations, but also to analyse issues in their ensuing large scale real-world deployments, which are now slowly starting to occur. This is needed as more traditional methods to handle this use case are difficult or impossible to use for QUIC [49]. To understand this, first consider that live deployment troubleshooting is typically done by two (sometimes overlapping) types of actors. Firstly, the ‘endpoint owners’, who manage load balancers, Content Delivery Network (CDN) nodes, origin servers, … Secondly, the ‘network operators’, who maintain the intermediate network infrastructure. For analysing traditional deployments, lower-layer tooling is prevalent (e.g., ipfix, netflow [23, 24]), yet to debug complex issues, insight at especially the transport layer is often needed. For TCP, most approaches again rely on capturing its wire image, typically at multiple locations, either at endpoints or from passive on-path measurements, and then use tools to find larger trends (e.g., tstat, caida, tranalyzer [19, 20, 64]). Core network health metrics, like latency and packet loss rate, are deduced from TCP metadata (e.g., by correlating TCP sequence and acknowledgement numbers) and used to localize network issues [18, 34, 50, 83].

Crucially however, these latter methods no longer work for QUIC, as it end-to-end encrypts most of this transport-level metadata [81], which is visible in plaintext on the wire for TCP and TLS. As such, to interpret QUIC traffic, it would either have to be decrypted live or be stored fully encrypted for post-hoc decryption and analysis. The latter could lead to huge storage requirements for QUIC, whereas for TCP ‘frame snapshots’ can be used [92], storing just the first 100 or so bytes of each packet. More problematic however, are the privacy and security implications of such a scheme. Storing the per-connection decryption keys undoes many of their ephemeral benefits and decrypting QUIC traffic not only reveals transport metadata, but also full application layer payloads. Adding insult to injury, to make this usable for network operators, endpoint owners would have to actively share the decryption keys or decrypted traces. One solution to this issue is to add additional unencrypted metadata to QUIC packet headers. This
has however traditionally been met with trepidation from the QUIC working group, and even the inclusion of a single ‘spin bit’ for latency measurement was heavily debated [2], with several parties indicating they will not support it [3]. It is unclear if similar ‘loss bits’ will make it into QUIC’s core [34].

Another solution, especially for the endpoint owners, could be to use a structured logging format like qlog. In contrast to the encrypted pcaps, these logs can easily be constructed to only contain the necessary metadata, saving on storage and bypassing many of the privacy-related issues. Additionally, they can contain internal application state, allowing for deeper root-cause analysis. Finally, as modern technologies like eBPF [36] allow extracting internal TCP state, we could even come full circle and also support this more powerful debugging approach for TCP(+TLS+H2). Yet, for all this to be feasible, we first need to determine if qlog indeed scales.

In the rest of this text, we address our three open questions, starting with qlog’s adoption and scalability in §2. We show a high investment from the QUIC community, with 12 out of 18 active QUIC stacks [4] currently implementing the format (notably including Cloudflare, Mozilla Firefox, Node.js and Facebook [5–8]). We use an expert survey among 28 implementers and researchers to gain additional insight into the reasons for this high uptake and into how the format and tools are used in practice. To confirm qlog indeed scales, we also report on an interview with Facebook engineers who use qlog and qvis at Internet scale to fine-tune their QUIC+H3 deployment for their mobile applications, and who log over 30 billion qlog events per day. We then discuss in §3 how tools like our qvis visualizations have been instrumental in debugging QUIC implementations. We show examples of issues encountered with multiplexing, packetization, congestion control and multipath extension design. We conclude in §4 that our approach indeed delivers on its potential, but that there is still a way to go towards practical use at scale. As we do not have enough space for appendices in this paper, all survey details, our source code, results, visualizations and other artefacts are made public on our websites [26, 59].

2 QLOG USAGE IN PRACTICE

We will not reiterate the full details of qlog’s format here, as they are described elsewhere [58, 61] and not really needed to understand this discussion. Instead, we focus on the reasons for and challenges discovered by qlog’s high uptake and use by the QUIC community. Per our survey, this broad adoption is driven by two main factors: firstly, the ability to use and create tools and visualizations (§3), and secondly, the flexibility of the format, which is leveraged in four main ways:

Firstly, there is the ability to easily define custom events. Each qlog event is defined by a timestamp, a category (e.g., “transport”), an event type (e.g., “packet_sent”) and some type-specific data (e.g., the size of the sent packet and its header fields). The qlog specification [61] mainly defines which categories, types and data layouts to use for typical QUIC+H3 events. Thanks to its use of JSON however, it can easily be modified or extended, which many developers have done for implementation-specific events. It also helped in debugging new QUIC features such as the ACK frequency, Datagram and Loss Bits extensions [35, 45, 69], as well as Multipath (§3.4). In qvis, most visualizations also show these custom events, preventing users from having to wait for a qlog or qvis update. Conversely, implementers can choose not to log certain event types (even those defined in the specification), allowing them to reduce file sizes or implement only the events they need.

Secondly, this flexibility goes beyond just QUIC+H3, as qlog can support additional network protocols. Some have already done this for DNS-over-QUIC [43], while others envision utilizing it for WebTransport, MASQUE and QUIC tunnel [71, 76, 88]. We ourselves have laid the groundwork for supporting TCP+TLS+H2 in qlog [9]. In this case, most of the basic events are obtained by transforming pcaps from (decrypted) pcap files into their qlog event counterparts. Fine-grained internal TCP state (e.g., congestion window, Round-Trip-Time (RTT) estimates) is retrieved by injecting eBPF probes inside the Linux kernel [36, 74], an approach that others have used as well [47, 82]. Both types of data are then spliced together into a single qlog trace. Eventually, we will also extract internal H2 state [1, 84], but even without this we have successfully analysed TCP+TLS+H2 deployments with the same tools we use for QUIC+H3 (see §3.1 - 3.3).

Thirdly, qlog files start with a protocol-agnostic “container” structure, which provides additional metadata on the file’s contents. This for example allows the aggregation of multiple distinct traces (e.g., from the client, the load balancer, the CDN node and the origin server) into a single qlog file, which can then be visualized in qvis’ sequence diagram (§3.4). This makes it easier to evaluate end-to-end behaviour in complex multi-tier setups, something which Facebook is experimenting with.

A final aspect is that qlog’s machine readable format allows other uses besides logging and visualizing. For example, some implementers utilize it as part of their (unit) testing pipeline, validating protocol behaviour by observing events in the qlog output [6, 51]. QUIC-Tracker and QUIC-Network-Simulator [72, 78] are also considering using it to verify interoperability testing results. Another example is Facebook, which stores all qlog events in a relational database. This allows them to easily query for traces with specific behaviour (e.g., high percentage of packet_lost events).

Knowing now why most QUIC implementers choose to support qlog, we should consider why some do not. In the survey, some large companies such as Google and Microsoft indicate their preference for an in-house format. Others hesitate assigning qlog a high priority, waiting for a student or intern
Figure 1: Multiplexing behaviour across three different QUIC stacks when downloading 10 1MB files in parallel. Each small colored rectangle is one payload frame belonging to a file. Black areas indicate which frames above them contain retransmitted data. Data arrives from left to right.

In contrast, even the compressed pcap exceeds 500MB, showing this alternative is indeed much more difficult to scale. Finally, note that even web-based tools like qvis easily scale to loading hundreds of MB of JSON.

3 VISUALIZATION CASE STUDIES

The behaviours and cross-layer interactions of network protocols are often complex and difficult to discern from textual logs. In contrast, even a simple graphical plot often provides immediate insight into a problem. Per our survey, one of the main reasons to use qlog is indeed the ability to both easily create custom visualizations, and re-use existing ones like our qvis toolsuite [55]. We have extended qvis substantially since its introduction, implementing five different powerful tools in over 10,000 lines of open source TypeScript code [54].

This section details how we and others have used tools to find bugs and inefficiencies in 16 QUIC+H3 stacks and five TCP+TLS+H2 implementations, and to validate our own improvements to the protocols. As interactive tools can be challenging to illustrate in screenshots, readers are encouraged to explore the discussed examples in qvis via our web page https://qlog.edm.uhasselt.be/anrw [59].

3.1 Stream multiplexing and prioritization

Modern protocol stacks often multiplex data from several parallel "streams" onto one connection (e.g., HTML, CSS and image files when loading a web page). This multiplexing can happen in various ways (e.g., files are sent sequentially as a whole or are scheduled via Round-Robin (RR) after being subdivided in chunks) and is typically steered using a prioritization system (e.g., H2’s dependency tree [15]). However, correctly implementing these systems is often hard in practice [63]. Data can end up misprioritized on the wire, potentially significantly degrading web page load performance [91].

The qvis multiplexing diagram (Figure 1) shows H2/H3’s payload carrying frames, appended on a horizontal line with coloring to discriminate the stream each belongs to. This immediately shows the multiplexing behaviour: RR schemes show frequent color changes (1, 2), while long contiguous...
Figure 2: Traces from five different TCP/QUIC servers visualized in the packetization diagram. The x-axis is in bytes received. Alternating colors on each row indicate the switch to a new TCP/QUIC packet, TLS record, QUIC/HTTP frame or HTTP stream. Elements that align vertically are packed into the lower layer’s payload.

swaths ③ mean sequential transfers. It is also easy to see abnormalities: ① has a sequential period at the start (which turned out to be due to unexpected interactions with flow control), while ③ unintentionally sent data in Last-In First-Out order, the worst-case for web performance [80] (this bug was subsequently fixed [42]). For QUIC, the diagram also highlights retransmissions. While lost TCP packets are always retransmitted with the highest priority in the original packet order, QUIC’s independent streams [46] give it more freedom. For example, while ③ is similar to TCP, ② instead treats retransmissions the same as normal data, and ① even changes its multiplexing behaviour entirely for lost data.

Our team has a long history of using visualizations to debug multiplexing. Before qvis, we discovered several H2 prioritization bugs in browsers [65, 91]. Later, Davies et al. used our early tools to find that only 9 out of 34 tested H2 deployments correctly implemented priorities [27]. We then researched viable alternate approaches [57], contributing to a redesign of priorities in H3 [66]. Currently, H3 developers employ qvis to debug their implementations of this new setup. Recently, we found that even after five years, four popular H2 servers still do not correctly support prioritization [59, 79], showing the importance of validating H3 stacks before wide deployment.

3.2 Packetization and framing

The multiplexing diagram in §3.1 was mainly concerned with the high-level ordering of HTTP DATA frames. Yet in the lower layers, these and other HTTP frames are subdivided into smaller protocol units for transport. H2 frames are packed in one or more TLS records, which are in turn distributed across TCP packets. QUIC foregoes TLS records, instead packing H3 frames in QUIC STREAM frames, and finally QUIC packets. How these units are sized and combined can significantly impact the protocol’s efficiency, as each subdivision adds some bytes of overhead (e.g., packet, record, and frame headers). Additionally, it can also carry security risks: if the edges of HTTP frames align directly with lower layer edges, attackers could in some cases derive HTTP resource sizes, usable in fingerprinting and CRIME-style attacks [31, 86].

The qvis packetization diagram (Figure 2) reveals the data packing particularities by vertically aligning each protocol layer’s units. It clearly distinguishes payloads from overhead (making the latter show up as vertical white areas) and alternates colors to show unit edges. The top row (“Stream IDs”) is similar to the multiplexing diagram (§3.1) to help identify how H2/H3 packages file data in DATA frames.

Zoomed out, this setup shows macro-level trends. For example, the blue H2 frames in A are the same size as the red TLS records they are packed in, as their edges align exactly. The connection also starts with small TLS records (hence the blurriness in the screenshot), but after about 1.1MB it switches to larger records (which are more efficient to encrypt [37, 38]). Additionally, each file (top row) ends on a much smaller TLS record, making file sizes easier to estimate by an attacker.

Zoomed in, lower level details can be discerned. In C, the HTTP layer strangely forces a flush of all outstanding data into a new TLS record whenever a new 9-byte H2 frame header is written. This is not only highly inefficient, it can also again reveal resource sizes. This behaviour is almost certainly an unknown server-side bug, as we have observed it on (and disclosed to) several high profile sites (e.g., Wikipedia in B).

Our stress test in C requests 1000 small files of 10 bytes each. We expect the implementation to bundle as many of them as possible in one packet, which it fails to do at times, unintentionally generating some tiny QUIC packets. This result caused the developers to revise their bundling logic. In D, a few smaller than average packets can be seen after approximately 10 and 30 sent QUIC packets. This is because according
to the specification [44], QUIC’s congestion window (cwnd) is not expressed in packets (as it is in TCP) but in bytes. D’s implementation aggressively fills its byte-allowance completely, even if that means generating smaller packets. Interestingly, our tests revealed that contrary to the specification, almost half of all QUIC implementations bypass this inefficiency and round up their cwnd to full-sized packets (similar to TCP). After we pointed this out to D’s developer, he decided to switch to this alternate approach as well [85].

In E we again see tiny packets, containing a single byte of H3 payload data. These off-by-one “errors” are induced because QUIC packet numbers (PNs) are variable-length encoded (values < 64 take up one byte, PNs < 16383 use two bytes, etc. [46]). This can be a problem as, before retransmission, lost QUIC packets are given a new, higher PN (to prevent the ACK ambiguity problem [39]). If this higher PN is encoded one byte larger than the previous, the updated packet can now exceed the maximum packet size [33], which is not allowed. Most QUIC implementations then fully re-frame the data but some, like E, decide the problem is rare enough and trade the occasional inefficiency (giving the leftover byte its own packet) for simpler code. This example shows that qvis can help surface and interpret even these esoteric behaviours.

For the future, examining packetization behaviour will be useful when adapting (new) application protocols to run over QUIC. Additionally, when using QUIC as an encrypted tunnel (e.g., MASQUE, QUIC tunnel [71, 76]), it can help identify and prevent inefficiencies, security issues and complex interactions (e.g., ensuring that QUIC and the tunneled protocol do not both run (conflicting) CC logic at the same time [40]).

### 3.3 Congestion control

Even after decades of evolution, congestion control approaches (CCs) are still a topic of active research and innovation [13, 89]. Bugs are still being found (e.g., Google found a decade-old bug in the Cubic CC when implementing QUIC [52]), CCs are still being fine tuned [75] and new CCs are being developed (e.g., COPA, BBRv2 [14, 22]). This is only expected to continue and even increase with QUIC, which is more open to experimentation than TCP due to its user-space implementations [52].

Yet, this experimentation might be stifled by the fact that the CC is one of the most complex components to implement correctly. This was also echoed in our survey, where debugging CCs was quoted as the main reason to create custom visualizations based on qlog. Several participants have created ad-hoc tools, implemented in a matter of minutes, to plot CC-related variables to observe their evolution over time.

qvis provides a more comprehensive congestion control graph, which can be seen as an extended combination of similar tcptrace tools [67]. The graph plots data sent and acknowledgements received on a timeline, as well as the congestion window, bytes in flight, connection and stream-level flow control limits and employed RTT measurements. Figure 3 shows part of the tool, emphasizing the difference in CC behaviour with and without pacing enabled. Pacing is the practice of spreading out packets across an RTT instead of sending them in short bursts, typically thought to reduce packet loss [12].

Using both qvis and custom tools, several bugs were found. As an example, Facebook diagnosed their BBR code not entering the probeRTT state at the right time. They also identified large-scale pacing issues between their transatlantic data centers due to errors in RTT measurement. Others mention finding bugs in QUIC’s retransmission logic during its complex handshake. During the development of PQUIC [29], we found the network emulation tool “mininet” to queue up an infinite amount of packets when using the default settings (ignoring the max_queue_size parameter). This was visible as a slow start phase and an ever-increasing RTT spanning the entire transfer. Additionally, we detected very short kernel freezes that were clearly visible as a narrow notch in the congestion graph. We also used the tool to validate our implementation of new CCs such as Westwood and BBR [21, 62] in PQUIC.

Finally, despite the complexity of this topic, qlog and qvis have made it possible for junior researchers and even bachelor students to make meaningful CC-related contributions (e.g., comparing QUIC’s loss detection and New Reno variant [44] with the Linux kernel’s TCP equivalent [74]). This highlights a broader advantage of our efforts: qlog and qvis can be used for teaching modern protocol behaviour in schools and companies, as also substantiated by our survey.
3.4 Multipath QUIC

One key feature of QUIC is its ability to migrate a connection from one set of IP addresses and ports to another [46]. Multipath QUIC (MP-QUIC) is a proposed extension improving this mechanism by allowing the simultaneous use of several network paths by modelling them as “uniflows” [25, 28]. An MP-QUIC implementation utilizes four new key components (see below), that each add significant complexity to QUIC’s packet sending and processing logic, and thus require insight into large amounts of internal state to properly debug.

To this end, when developing the Multipath extension in our Pluginized QUIC (PQUIC) implementation [29], we have leveraged qlog’s flexibility and substantially extended it. We added custom qlog events for the new multipath-related state and also re-scope existing connection-level events to the path level by adding the “uniflow ID”. Subsequently, as most of the qvis tools available today were still under development at that time, we developed two custom multipath-enabled tools. The first is a simplified version of qvis’ sequence diagram (Figure 4). It shows packet loss and reordering, and uses colored arrows to indicate the separate uniflows. A packet’s full content can be viewed by hovering over its summary text. The second is a general timeline tool allowing the visualization of any qlog event data, selected through the use of a simple grammar. It for instance allows visualizing the congestion control variables per path and over time, providing similar functionality to mptcptrace [41]. We have used these two tools to debug the four key MP-QUIC components:

First, the path manager, which decides which potential network paths are actually usable at any time by using a link failure detection heuristic. By matching the time at which our experiments triggered a link failure with the time we observe the path manager retiring a uniflow and probing another one, we detected and corrected false positives in this heuristic.

Second, the packet scheduler, which decides which of several uniflows should be chosen for sending a packet in the tool, we saw our packet scheduler re-injecting packets into uniflows that were not validated after a link failure. The per-uniflow colors revealed these issues in a matter of seconds.

Third, the frame scheduler, which composes a packet from a series of frames based on the selected uniflow. By observing the Stream IDs of STREAM frames sent on each path using the timeline tool, we were able to quickly assess the correctness of our path-aware frame scheduler.

Fourth, the congestion controller, which was adapted to retain separate state for each network path. By plotting the RTT estimates for each path in an asymmetric path scenario, we found MP_ACK frames being incorrectly handled. The timeline plot clearly exhibited two modes in the raw RTT estimates, showing signals from both paths being mixed together. This finding partly motivated the change from bidirectional paths to uniflows in the latest MP-QUIC design [25].

These results show qlog’s potential to help design and realize the complex new features envisioned in QUIC’s future.

4 CONCLUSION & CHALLENGES

With this work, we feel we have adequately answered two of the three open questions listed in the introduction. Firstly, it is clear that a substantial part of the QUIC community (including Facebook, Mozilla, and Cloudflare) has adopted qlog and qvis, both for debugging initial implementations (100% of participants) and for supporting larger (planned) deployments (60% of participants from large companies).

Secondly, what we discussed in this work was only a small selection of the results and insights we and others obtained from using qlog and qvis. Still, we hope the reader will agree that these examples show that our approach indeed radically improves the ability to implement, debug and evolve not just QUIC+H3, but other protocols as well. Additionally, we feel the ability to easily observe QUIC’s complex behaviour will be central in bringing it to the wider masses, helping to educate newcomers, as well as facilitating further academic research.

The last question, whether the structured endpoint logging approach could scale and replace packet captures in large deployments, has however only been partially answered. We feel we have shown that it is indeed feasible for endpoint owners, as also evidenced by Facebook’s deployment. We also intend to add changes (§2) to qlog to make it easier, though their effectiveness will also have to be proven. However, it is difficult to see how this can work directly for network operators, as they lack the ability to easily produce qlogs. Potentially, endpoint owners could share (aggregated/limited versions of) qlogs with these intermediaries, but this would require additional infrastructure for log transport, storage, aggregation and, most importantly, access control and privacy assurance. As such, we hope our results will aid further IETF discussion on comparing our approach with the alternative of adding additional plaintext metadata to QUIC’s packet header for the network operator use case. Finally, we hope the IETF can help us explore applying qlog to more protocols.
ACKNOWLEDGMENTS

Robin Marx is a SB PhD fellow at FWO, Research Foundation Flanders, #1S02717N. Maxime Piraux’s contribution is partially supported by funding from the Walloon Government (DG06) within the MQUIC project. The authors would like to thank Olivier Bonaventure, Lucas Pardue, Wim Lom- otte, Mariano Di Martino, Joris Herbots, Wouter Vannomtort, Jens Bruggemans, Pallieter Verlinden, Jimmy Cleuren, and Quentin De Coninck for their help in reviewing earlier versions of this work and Jonas Reynders, Jeremy Lainé, Marten Seemann, Christian Huitema, Matt Joras, Nick Banks, Jana Iyengar, Mirja Kühlwein, and many other members of the QUIC community for their efforts on this topic in general.

REFERENCES


Debugging Modern Web Protocols with qlog


[86] Mathy Vanhoef and Tom Van Goethem. 2016. HEIST: HTTP Encrypted Information can be Stolen through TCP-windows. In Black Hat US Briefings, Location: Las Vegas, USA.


