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+H2 stack (e.g., wireshark, tcptrace, and captcp [10, 67, 70]), 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 packets 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

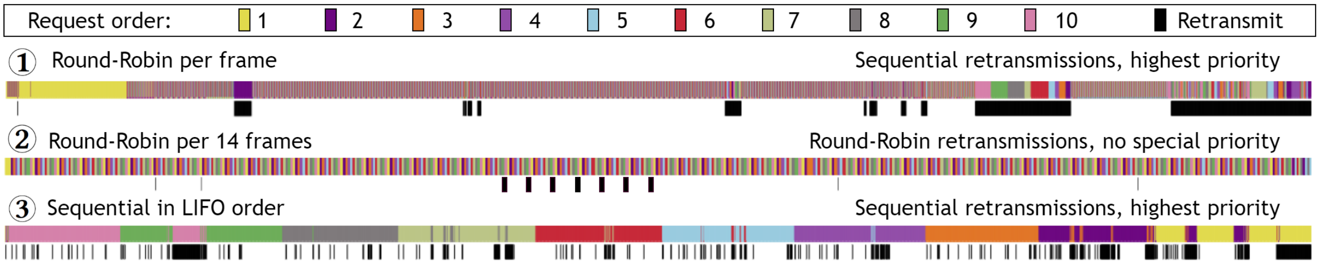

Figure 1: Multiplexing behaviour across three different QUIC stacks when downloading 10 1MB files in parallel. Each small colored rectangle is one payload frame belonging to a file. Black areas indicate which frames above them contain retransmitted data. Data arrives from left to right.

developer or on the availability of libraries [56, 68], despite also indicating they suffer from the lack of additional debuggability. They also fear that both the initial implementation and its later maintenance entail a considerable time investment. This is somewhat contradicted by our own experiences: our *qlog* integration in PQUIC [29] is isolated and made flexible so that additional plugins can easily inject new events (§3.4).

A final argument heard against *qlog* is that its use of JSON might not scale [77]. We chose JSON because it is flexible, has excellent built-in support in most programming environments, and allows plaintext search. However, especially larger companies fear the format is too verbose (leading to large file sizes) and too slow to (de)serialize to use in production. They advocate using a more optimized binary format [30, 87], even though these typically lack many of JSON's benefits. Facebook, the only party with experience deploying *qlog* at scale, posits a more nuanced view. They find *qlog* is indeed two to three times as large, and takes 50% longer to serialize, than their previous in-house binary format. However, this overhead is manageable on the server-side. They *qlog* close to 10% of all QUIC connections, selected via random sampling, scaling to over 30 billion daily *qlog* events [11]. Contrarily, on the client-side, the large size does often prevent them from uploading full-length *qlogs* via the users' often constrained cellular links. Still, they would not want to move to a binary format if it meant losing flexibility and feel the CPU overhead can be reduced by developing a *qlog*-specific JSON serializer.

It is clear that *qlog* would benefit from a solution which balances flexibility and efficiency. After evaluating several options [53, 77], we settled on adding a two-pronged "optimized mode". Firstly, we employ logical compression by replacing repeated values with an index into a dynamic dictionary (akin to H3's QPACK [48]). Secondly, we use CBOR [17] to encode this smaller *qlog* and its dictionary. CBOR is JSON's direct binary counterpart, compresses well and retains flexibility [32, 73]. For reference, the *qlog* file for a 500MB download is normally 276MB, but only 91MB in optimized mode, which easily compresses down to 15MB, while the optimized (but less flexible) binary protobuf equivalent [30] ends at a compressed 14MB. 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

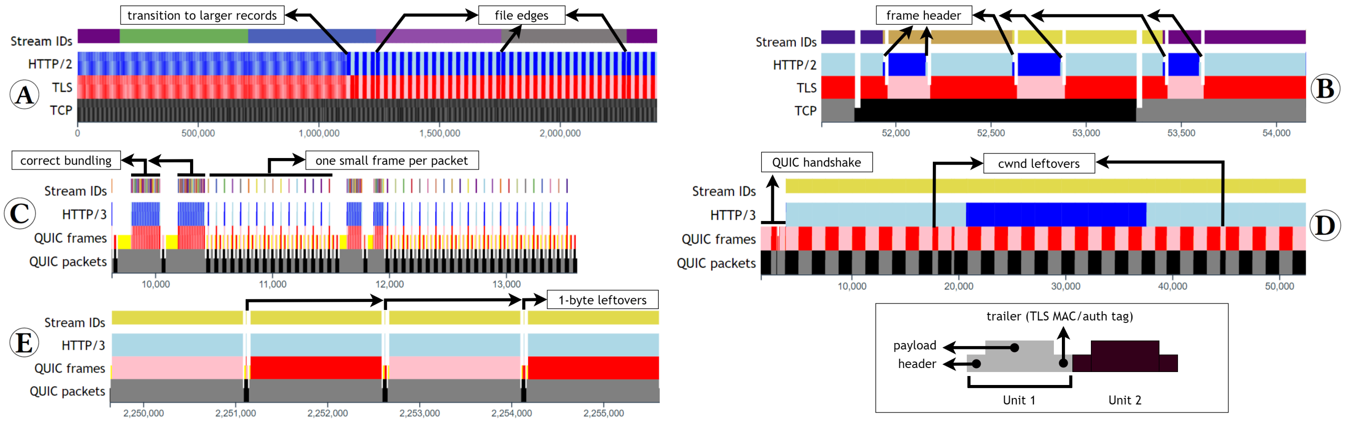

Figure 2: Traces from five different TCP/QUIC servers visualized in the packetization diagram. The x-axis is in bytes received. Alternating colors on each row indicate the switch to a new TCP/QUIC packet, TLS record, QUIC/HTTP frame or HTTP stream. Elements that align vertically are packed into the lower 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 Ⓐ 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 Ⓑ, 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 Ⓑ).

Our stress test in Ⓒ 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 Ⓓ, a few smaller than average packets can be seen after approximately 10 and 30 sent QUIC packets. This is because according

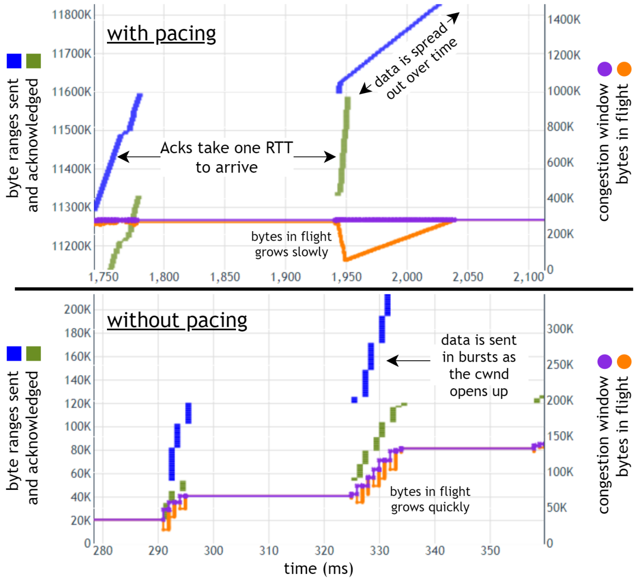

Figure 3: Detail of the congestion control graph.

to the specification [44], QUIC's congestion window (cwnd) is not expressed in packets (as it is in TCP) but in bytes. Ⓓ'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 Ⓓ's developer, he decided to switch to this alternate approach as well [85].

In Ⓔ 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 Ⓔ, 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-scoped 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].

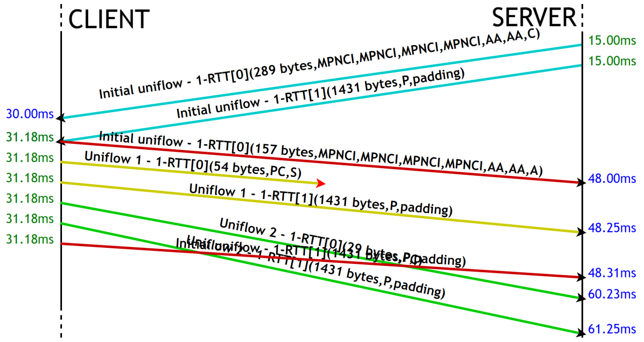
CLIENT
SERVER
15.00ms
15.00ms
Initial uniflow - 1-RTT[0](289 bytes,MPNCI,MPNCI,MPNCI,MPNCI,AA,AA,C)
Initial uniflow - 1-RTT[1](1431 bytes,P,padding)
30.00ms
31.18ms
Initial uniflow - 1-RTT[0](157 bytes,MPNCI,MPNCI,MPNCI,MPNCI,AA,AA,A)
31.18ms
Uniflow 1 - 1-RTT[0](54 bytes,PC,S)
31.18ms
Uniflow 1 - 1-RTT[1](1431 bytes,P,padding)
31.18ms
31.18ms
Uniflow 2 - 1-RTT[0](29 bytes,P,padding)
31.18ms
Initial uniflow - 1-RTT[1](1431 bytes,P,C)
Uniflow 2 - 1-RTT[1](1431 bytes,P,padding)
48.00ms
48.25ms
48.31ms
60.23ms
61.25ms

Figure 4: Sequence diagram of a partial MP-QUIC uniflow establishment, with reordering and packet loss.

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 ensurance. 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 (DGO6) within the MQUIC project. The authors would like to thank Olivier Bonaventure, Lucas Pardue, Wim Lamotte, Mariano Di Martino, Joris Herbots, Wouter Vanmontfort, 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 Kühlewind, and many other members of the QUIC community for their efforts on this topic in general.

REFERENCES

[1] 2012. NetLog: Chrome's network logging system. https://www.chromium.org/developers/design-documents/network-stack/netlog.

[2] 2018. IETF 101 - spin bit discussion. https://github.com/quicwg/wg-materials/blob/master/ietf101/minutes.md.

[3] 2018. IETF 103 - spin bit discussion. https://github.com/quicwg/wg-materials/blob/master/ietf103/minutes.md.

[4] 2020. Active QUIC implementations. https://github.com/quicwg/base-drafts/wiki/Implementations.

[5] 2020. Cloudflare quiche. https://github.com/cloudflare/quiche.

[6] 2020. Facebook mvfst. https://github.com/facebookincubator/mvfst.

[7] 2020. Mozilla neqo. https://github.com/mozilla/neqo.

[8] 2020. Node.js QUIC. https://github.com/nodejs/quic.

[9] 2020. qlog for TCP+TLS+HTTP/2 proof of concept. https://github.com/quiclog/qvis/blob/master/visualizations/src/components/filemanager/pcapconverter/qlog_tcp_tls_h2.ts.

[10] 2020. Wireshark. https://www.wireshark.org/.

[11] Lior Abraham, John Allen, Oleksandr Barykin, Vinayak Borkar, Bhuwan Chopra, Ciprian Gerea, Daniel Merl, Josh Metzler, David Reiss, Subbu Subramanian, et al. 2013. Scuba: diving into data at facebook. *Proceedings of the VLDB Endowment* 6, 11 (2013), 1057–1067.

[12] Amit Aggarwal, Stefan Savage, and Thomas Anderson. 2000. Understanding the performance of TCP pacing. In *Proceedings IEEE INFOCOM 2000. Conference on Computer Communications. Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies.* IEEE, 1157–1165.

[13] Rasool Al-Saadi, Grenville Armitage, Jason But, and Philip Branch. 2019. A survey of delay-based and hybrid TCP congestion control algorithms. *IEEE Communications Surveys & Tutorials* 21, 4 (2019), 3609–3638.

[14] Venkat Arun and Hari Balakrishnan. 2018. Copa: Practical delay-based congestion control for the internet. In *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18).* 329–342.

[15] M. Belshe, R. Peon, and M. Thomson. 2015. *Hypertext Transfer Protocol Version 2 (HTTP/2).* RFC 7540. RFC Editor. http://www.rfc-editor.org/rfc/rfc7540.txt

[16] Mike Bishop. 2020. *Hypertext Transfer Protocol Version 3 (HTTP/3).* Internet-Draft draft-ietf-quic-http-27. IETF Secretariat. http://www.ietf.org/internet-drafts/draft-ietf-quic-http-27.txt

[17] C. Bormann and P. Hoffman. 2013. *Concise Binary Object Representation (CBOR).* RFC 7049. RFC Editor.

[18] Fabio Bulgarella, Mauro Cociglio, Giuseppe Fioccola, Guido Marchetto, and Riccardo Sisto. 2019. Performance measurements of QUIC communications. In *Proceedings of the Applied Networking Research Workshop.* 8–14.

[19] Stefan Burschka and Benoît Dupasquier. 2016. Tranalyzer: Versatile high performance network traffic analyser. In *2016 IEEE Symposium Series on Computational Intelligence (SSCI).* IEEE, 1–8.

[20] CAIDA. 2020. Overview of CAIDA Software Tools. https://www.caida.org/tools/.

[21] Neal Cardwell, Yuchung Cheng, Soheil Yeganeh, and Van Jacobson. 2017. *BBR Congestion Control.* Internet-Draft draft-cardwell-iccrg-bbr-congestion-control-00. IETF Secretariat. http://www.ietf.org/internet-drafts/draft-cardwell-iccrg-bbr-congestion-control-00.txt

[22] N Cardwell, Yuchung Cheng, S Hassas Yeganeh, Ian Swett, Victor Vasiliev, Priyaranjan Jha, Yousuk Seung, Matt Mathis, and Van Jacobson. 2019. BBRv2: A Model-Based Congestion Control. In *Presentation in ICCRG at IETF 104th meeting.*

[23] B. Claise. 2004. *Cisco Systems NetFlow Services Export Version 9.* RFC 3954. RFC Editor.

[24] B. Claise, B. Trammell, and P. Aitken. 2013. *Specification of the IP Flow Information Export (IPFIX) Protocol.* RFC 7011. RFC Editor.

[25] Quentin De Coninck, François Michel, and Olivier Bonaventure. 2020. *Multipath Extensions for QUIC (MP-QUIC).* Internet-Draft draft-deconinck-quic-multipath-04. IETF Secretariat. http://www.ietf.org/internet-drafts/draft-deconinck-quic-multipath-04.txt

[26] Quentin De Coninck, Maxime Piraux, and Olivier Bonaventure. 2020. Pluginized QUIC. https://pquic.org.

[27] Andy Davies and Patrick Meenan. 2018. Tracking HTTP/2 Prioritization Issues. https://github.com/andydavies/http2-prioritization-issues.

[28] Quentin De Coninck and Olivier Bonaventure. 2017. Multipath QUIC: Design and Evaluation. In *Proceedings of the 13th International Conference on emerging Networking EXperiments and Technologies.* ACM, 160–166.

[29] Quentin De Coninck, François Michel, Maxime Piraux, Florentin Rochet, Thomas Given-Wilson, Axel Legay, Olivier Pereira, and Olivier Bonaventure. 2019. Pluginizing QUIC. In *Proceedings of the ACM Special Interest Group on Data Communication.* ACM, 59–74.

[30] Google Developers. 2020. Protocol Buffers. https://developers.google.com/protocol-buffers.

[31] Mariano Di Martino, Peter Quax, and Wim Lamotte. 2019. Realistically Fingerprinting Social Media Webpages in HTTPS Traffic. In *Proceedings of the 14th International Conference on Availability, Reliability and Security (ARES '19).* 10. https://doi.org/10.1145/3339252.3341478

[32] J. Dickinson, J. Hague, S. Dickinson, T. Manderson, and J. Bond. 2019. *Compacted-DNS (C-DNS): A Format for DNS Packet Capture.* RFC 8618. RFC Editor.

[33] Godred Fairhurst, Tom Jones, Michael Tuexen, Irene Ruengeler, and Timo Voelker. 2020. *Packetization Layer Path MTU Discovery for Datagram Transports.* Internet-Draft draft-ietf-tsvwg-datagram-plpmtud-08. IETF Secretariat. http://www.ietf.org/internet-drafts/draft-ietf-tsvwg-datagram-plpmtud-08.txt

[34] A. Ferrieux, I. Hamchaoui, I. Lubashev, and D. Tikhonov. 2020. *Packet Loss Signaling for Encrypted Protocols.* Internet-Draft draft-ferrieuxhamchaoui-quic-lossbits-03. IETF Secretariat. https://tools.ietf.org/id/draft-ferrieuxhamchaoui-quic-lossbits-03.txt

[35] Alexandre Ferrieux, Isabelle Hamchaoui, Igor Lubashev, and Dmitri Tikhonov. 2020. *Packet Loss Signaling for Encrypted Protocols.* Internet-Draft draft-ferrieuxhamchaoui-quic-lossbits-03. IETF Secretariat. http://www.ietf.org/internet-drafts/draft-ferrieuxhamchaoui-quic-lossbits-03.txt

[36] Matt Fleming. 2017. A thorough introduction to eBPF. *Linux Weekly News* (December 2017). https://old.lwn.net/Articles/740157/.

[37] John Graham-Cumming. 2016. Optimizing TLS over TCP to reduce latency. https://blog.cloudflare.com/optimizing-tls-over-tcp-to-reduce-latency.

[38] Ilya Grigorik. 2013. Optimizing TLS Record Size and Buffering Latency. https://www.igvita.com/2013/10/24/optimizing-tls-record-size-and-buffering-latency.

[39] Andrei Gurtov and Sally Floyd. 2004. Resolving acknowledgment ambiguity in non-SACK TCP. *Next Generation Teletraffic and Wired/Wireless Advanced Networking (NEW2AN'04)* (2004).

[40] Russell Harkanson, Yoohwan Kim, Ju-Yeon Jo, and Khanh Pham. 2019. Effects of TCP Transfer Buffers and Congestion Avoidance Algorithms on the End-to-End Throughput of TCP-over-TCP Tunnels. In *16th International Conference on Information Technology-New Generations (ITNG 2019)*. Springer, 401–408.

[41] Benjamin Hesmans and Olivier Bonaventure. 2014. Tracing multipath TCP connections. In *Proceedings of the 2014 ACM Conference on SIGCOMM*. 361–362.

[42] Christian Huitema. 2020. Files are being sent LIFO. https://github.com/private-octopus/picoquic/issues/768.

[43] Christian Huitema, Melinda Shore, Allison Mankin, Sara Dickinson, and Jana Iyengar. 2019. *Specification of DNS over Dedicated QUIC Connections*. Internet-Draft draft-huitema-quic-dnsoquic-06. IETF Secretariat. http://www.ietf.org/internet-drafts/draft-huitema-quic-dnsoquic-06.txt

[44] Jana Iyengar and Ian Swett. 2020. *QUIC Loss Detection and Congestion Control*. Internet-Draft draft-ietf-quic-recovery-27. IETF Secretariat. http://www.ietf.org/internet-drafts/draft-ietf-quic-recovery-27.txt

[45] Jana Iyengar and Ian Swett. 2020. *Sender Control of Acknowledgement Delays in QUIC*. Internet-Draft draft-iyengar-quic-delayed-ack-00. IETF Secretariat. http://www.ietf.org/internet-drafts/draft-iyengar-quic-delayed-ack-00.txt

[46] Jana Iyengar and Martin Thomson. 2020. *QUIC: A UDP-Based Multiplexed and Secure Transport*. Internet-Draft draft-ietf-quic-transport-27. IETF Secretariat. http://www.ietf.org/internet-drafts/draft-ietf-quic-transport-27.txt

[47] Keertan Kini. 2017. *Vessel: a lightweight container for network analysis*. Ph.D. Dissertation. Massachusetts Institute of Technology.

[48] Charles Krasic, Mike Bishop, and Alan Frindell. 2020. *QPACK: Header Compression for HTTP/3*. Internet-Draft draft-ietf-quic-qpack-14. IETF Secretariat. http://www.ietf.org/internet-drafts/draft-ietf-quic-qpack-14.txt

[49] Mirja Kuehlewind and Brian Trammell. 2020. *Manageability of the QUIC Transport Protocol*. Internet-Draft draft-ietf-quic-manageability-06. IETF Secretariat. http://www.ietf.org/internet-drafts/draft-ietf-quic-manageability-06.txt

[50] Mirja Kühlewind, Tobias Bühler, Brian Trammell, Stephan Neuhaus, Roman Müntener, and Gorry Fairhurst. 2017. A path layer for the Internet: Enabling network operations on encrypted protocols. In *2017 13th International Conference on Network and Service Management (CNSM)*. IEEE, 1–9.

[51] Jeremy Lainé. 2020. aioquic. https://github.com/aiortc/aioquic.

[52] Adam Langley, Alistair Riddoch, Alyssa Wilk, Antonio Vicente, Charles Krasic, Dan Zhang, Fan Yang, Fedor Kouranov, Ian Swett, Janardhan Iyengar, et al. 2017. The quic transport protocol: Design and internet-scale deployment. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. 183–196.

[53] Robin Marx. 2020. qlog format converters. https://github.com/quiclog/pcap2qlog/tree/binary/src/converters.

[54] Robin Marx. 2020. qvis toolsuite code. https://github.com/quiclog/qvis.

[55] Robin Marx. 2020. qvis toolsuite live. https://qvis.edm.uhasselt.be.

[56] Robin Marx. 2020. TypeScript qlog implementation. https://github.com/quiclog/qlog/tree/master/TypeScript.

[57] Robin Marx., Tom De Decker., Peter Quax., and Wim Lamotte. 2019. Of the Utmost Importance: Resource Prioritization in HTTP/3 over QUIC. In *Proceedings of the 15th International Conference on Web Information Systems and Technologies - Volume 1: WEBIST,*. INSTICC, SciTePress, 130–143. https://doi.org/10.5220/0008191701300143

[58] Robin Marx, Wim Lamotte, Jonas Reynders, Kevin Pittevils, and Peter Quax. 2018. Towards QUIC debuggability. In *Proceedings of the Workshop on the Evolution, Performance, and Interoperability of QUIC*. 1–7.

[59] Robin Marx and Maxime Piraux. 2020. Artefacts for this paper. https://qlog.edm.uhasselt.be/anrw.

[60] Robin Marx and Kevin Pittevils. 2019. quicker, a QUIC implementation in typescript. https://github.com/rmarx/quicker.

[61] Robin Marx, Marten Seemann, and Jeremy Lainé. 2019. The IETF I-D documents for the qlog format. https://github.com/quiclog/internet-drafts.

[62] Saverio Mascolo, Claudio Casetti, Mario Gerla, Medy Y Sanadidi, and Ren Wang. 2001. TCP westwood: Bandwidth estimation for enhanced transport over wireless links. In *Proceedings of the 7th annual international conference on Mobile computing and networking*. 287–297.

[63] Patrick Meenan. 2019. Better HTTP/2 Prioritization for a Faster Web. https://blog.cloudflare.com/better-http-2-prioritization-for-a-faster-web/.

[64] A Finamore M Mellia M Meo, MM Munafo, and D Rossi. 2020. 10-year Experience of Internet Traffic Monitoring with Tstat. (2020).

[65] Daan De Meyer. 2018. h2vis: tools for HTTP/2. https://github.com/DaanDeMeyer/h2vis.

[66] Kazuho Oku and Lucas Pardue. 2020. *Extensible Prioritization Scheme for HTTP*. Internet-Draft draft-ietf-httpbis-priority-00. IETF Secretariat. http://www.ietf.org/internet-drafts/draft-ietf-httpbis-priority-00.txt

[67] Shawn Ostermann. 2005. Tcptrace.

[68] Lucas Pardue. 2020. qlog Rust crate. https://crates.io/crates/qlog.

[69] Tommy Pauly, Eric Kinnear, and David Schinazi. 2020. *An Unreliable Datagram Extension to QUIC*. Internet-Draft draft-ietf-quic-datagram-00. IETF Secretariat. http://www.ietf.org/internet-drafts/draft-ietf-quic-datagram-00.txt

[70] Hagen Paul Pfeifer. 2013. Captcp. http://research.protocollabs.com/captcp/.

[71] Maxime Piraux and Olivier Bonaventure. 2020. *Tunneling Internet protocols inside QUIC*. Internet-Draft draft-piraux-quic-tunnel-01. IETF Secretariat. http://www.ietf.org/internet-drafts/draft-piraux-quic-tunnel-01.txt

[72] Maxime Piraux, Quentin De Coninck, and Olivier Bonaventure. 2018. Observing the evolution of QUIC implementations. In *Proceedings of the Workshop on the Evolution, Performance, and Interoperability of QUIC*. 8–14.

[73] Shahid Raza, Joel Hoeglund, Goeran Selander, John Mattsson, and Martin Furuhed. 2019. *CBOR Profile of X.509 Certificates*. Internet-Draft draft-raza-ace-cbor-certificates-03. IETF Secretariat. http://www.ietf.org/internet-drafts/draft-raza-ace-cbor-certificates-03.txt

[74] Jonas Reynders. 2020. QUICSim. https://github.com/moonfalir/quicSim-docker.

[75] Jan Rüth, Ike Kunze, and Oliver Hohlfeld. 2019. An empirical view on content provider fairness. In *2019 Network Traffic Measurement and Analysis Conference (TMA)*. IEEE, 177–184.

[76] David Schinazi. 2020. *The MASQUE Protocol*. Internet-Draft draft-schinazi-masque-protocol-01. IETF Secretariat. http://www.ietf.org/internet-drafts/draft-schinazi-masque-protocol-01.txt

[77] Marten Seemann. 2019. Consider moving qlog to a binary format. https://github.com/quiclog/internet-drafts/issues/30.

[78] Marten Seemann and Jana Iyengar. 2020. Network Simulator for QUIC benchmarking. https://github.com/marten-seemann/quic-network-simulator.

[79] James Snell. 2020. Node.js does not support HTTP/2 priorities. https://twitter.com/jasnell/status/1245410283582918657.

[80] Ian Swett and Robin Marx. 2019. HTTP Priority design team update - IETF 107. https://github.com/httpwg/wg-materials/blob/gh-pages/

ietf106/priorities.pdf.

[81] Martin Thomson and Sean Turner. 2020. *Using TLS to Secure QUIC*. Internet-Draft draft-ietf-quic-tls-27. IETF Secretariat. http://www.ietf.org/internet-drafts/draft-ietf-quic-tls-27.txt

[82] Olivier Tilmans and Olivier Bonaventure. 2019. COP2: Continuously Observing Protocol Performance. *arXiv preprint arXiv:1902.04280* (2019).

[83] B. Trammell and M. Kuehlewind. 2018. *The QUIC Latency Spin Bit*. Internet-Draft draft-ietf-quic-spin-exp-01. IETF Secretariat. https://tools.ietf.org/id/draft-ietf-quic-spin-exp-01.txt

[84] Tatsuhiro Tsujikawa. 2015. Nghttp2: HTTP/2 C library and tools. https://github.com/nghttp2/nghttp2.

[85] Tatsuhiro Tsujikawa. 2020. Round up cwnd left to the maximum UDP packet size. https://github.com/ngtcp2/ngtcp2/commit/0a28514bbbb37d85dc6e2622357687166669192a.

[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*.

[87] Kenton Varda. 2020. Cap'n Proto. https://capnproto.org/.

[88] Victor Vasiliev. 2019. *The WebTransport Protocol Framework*. Internet-Draft draft-vvv-webtransport-overview-01. IETF Secretariat. http://www.ietf.org/internet-drafts/draft-vvv-webtransport-overview-01.txt

[89] Ranysha Ware, Matthew K Mukerjee, Srinivasan Seshan, and Justine Sherry. 2019. Beyond Jain's Fairness Index: Setting the Bar For The Deployment of Congestion Control Algorithms. In *Proceedings of the 18th ACM Workshop on Hot Topics in Networks*. 17–24.

[90] QUIC wg. 2020. QUIC Working Group adopted documents. https://datatracker.ietf.org/wg/quic/documents/.

[91] Maarten Wijnants, Robin Marx, Peter Quax, and Wim Lamotte. 2018. HTTP/2 Prioritization and Its Impact on Web Performance. In *Proceedings of the 2018 World Wide Web Conference* (Lyon, France) *(WWW '18)*. 1755–1764. https://doi.org/10.1145/3178876.3186181

[92] Wireshark. 2010. Frame snapshot length. https://wiki.wireshark.org/SnapLen.