Quicly now! Go take two pieces of Quiche to Quinn!

A Study of QUIC and HTTP/3 Implementation Diversity

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

Joris Herbots
Wim Lamotte
Hasselt University – tUL – EDM
Diepenbeek, Belgium
{first.last}@uhasselt.be

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

	aioquic	google	lsquic	mvfst	ngtcp2	picoquic	quic-go	quiche	quicly	quinn
Flow Control category (FC)	2	1	1	1	1	2	1	1	1	1
Multiplexing scheduler	SEQ	RR	RR	RR	SEQ	SEQ	RR	RR	RR	RR
Retransmission approach (RA)	2	1	2	3	2	2	2	1	2	2
0 RTT approach (ZR)	1	1	2	3	1	2	2	1	2	1
DATA frame size	large	medium	small	large	small	large	large	small	large	small
Worst case packetization goodput efficiency	90.34%	95.02%	92.54%		90.88%	87.94%			91.52%	83.92%
Dynamic packet sizing (PMTUD)	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗
Acknowledgment frequency (#packets)	1	2-10	2-8	10	2-4	2-6	2-9	1	2	
Congestion Control (CC) New Reno | Cubic | BBR	✓|✗|✗	✗|✓|✓	✗|✓|✓	✓|✓|✓	✓|✗|✗	✓|✓|✓	✗|✓|✗	✓|✓|✗	✓|✗|✗	✓|✗|✗

Table 1: Selective behavioral comparison of 10 prevailing IETF QUIC implementations Empty slots indicate we do not have results for this data point. SEQ = Sequential, RR = Round-Robin

ABSTRACT

The QUIC and HTTP/3 protocols are quickly maturing together with their implementations, though many of their low-level behaviours are not yet well-understood. To help improve this, we empirically compare 15 IETF QUIC/H3 implementations for advanced features like Flow and Congestion Control, 0-RTT, Multiplexing, and Packetization. We find a large heterogeneity between stacks, discuss uncovered bugs and conclude that most implementations are not fully optimized or validated yet. We argue that future work must prioritize rigorous root-cause analysis of observed behaviours, and show this is possible by employing our qlog and qvis tools.

1 INTRODUCTION & MOTIVATION

In 2020, after nearly four years, the new QUIC and HTTP/3 (H3) protocol specifications [7, 24] are finally nearing completion. This long period is a testament to their complexity, as they combine decades of best practices, lessons learned from TCP, SCTP and HTTP/2 (H2), and advanced new features (like zero Round-Trip-Time (RTT) connection establishment) into a new Web protocol suite. To help verify that the protocols' design choices actually hold up in practice and to prepare for deployment, several parties have been continuously updating over 18 different QUIC/H3 implementations [2]. These stacks are regularly tested on their so-called "interoperability", whereby clients from one implementation test features of servers from other codebases. This is done both manually and automatically in projects such as QUIC Tracker and QUIC Interop Runner [3, 35]. Despite this, bugs are still regularly uncovered (several by our research) and the more advanced features are often not yet well supported or finetuned.

This is partly because existing tests mainly consider compatibility of the protocols' binary wire image and the mandatory parts of the specifications (i.e., MUST and MUST NOT). There are however many protocol features and situations

for which the guidelines are much less clear and up to the developer's choice. These features, such as Flow Control, Congestion Control, Data Multiplexing, Packetization, and 0 RTT are often more difficult to evaluate in an automated fashion, yet arguably can have a large impact on protocol performance and behaviour. A good example of this can be found in H2's highly complex Prioritization setup [41], which controls how bandwidth is distributed across concurrent Web page resource downloads (§3.2). This system was added late in H2's design and poorly validated prior to deployment. Consequently, even today, 5 years after the protocol's standardization, many H2 servers and clients do not properly support this feature [13, 30, 41], and it was decided to fully redesign it for H3 [32]. As such, we feel it is imperative to evaluate implementations of these more loosely defined QUIC features.

While there is some prior academic work that evaluates some of these aspects for QUIC [8, 11, 31], most of it is older and outdated, as it considers Google's initial QUIC version (gQUIC) [17]. While gQUIC and IETF QUIC are similar in concepts, their implementation details are fundamentally divergent. Furthermore, several critical evaluations have shown that some research lacked scientific rigor and root-cause analysis, often misconfiguring their evaluated implementations, and potentially reaching erroneous conclusions [16, 18]. We additionally believe that the relatively low interest [33, 34, 37] of the academic community in IETF QUIC since the early research is due to the protocol's complexity and rapid evolution/unstableness. We however identified this problem early, recognizing the need for advanced QUIC debugging and analysis techniques. To this end, in 2018 we proposed qlog, a standardized endpoint logging format, and the accompanying qvis visualizations and tools [26, 28, 29]. Both qlog and qvis have found broad uptake in the IETF QUIC community since, with 12/18 implementations outputting qlog, and qvis providing advanced analysis tools for several protocol features.

As such, as the protocols, their implementations and our tools are reaching maturity, we feel it is now finally time to evaluate if they are ready to be deployed, researched and evaluated. In this work, we use qlog and qvis to assess implementation differences between and maturity of 15 of the 18 active IETF QUIC and H3 implementations. Our results for 10 of those stacks are summarized in Table 1. As other previously mentioned projects target interoperability testing [3, 35], we instead focus on protocol aspects that are difficult to automatically measure and that are expected to have a high measure of heterogeneity across implementations (§3). We indeed identify large differences between implementations and find that many advanced features are not yet tuned or validated in many stacks. Still, we conclude that with the powerful qlog and qvis tooling, proper analysis of implementation behaviour is possible, and thus researchers can start evaluating QUIC.

2 EXPERIMENTAL METHODOLOGY

To deeply evaluate 15 QUIC implementations in a manageable amount of time, we rely heavily on the structured qlog format [29]. As 12 QUIC stacks output qlog, it is feasible to have always at least one end of a cross-implementation connection outputting this format (and often both). We then both automatically process qlogs with scripts, and evaluate them manually via the qvis visualizations [26]. We use two main qlog sources: firstly, the QUIC interop runner tests [3], which employ an ns-3 network emulation setup between dockerized versions of 10 different QUIC stacks. While these tests do not explicitly consider our targeted behaviours, some of them can be used to obtain the insights we require (e.g., concurrent file transfer tests also allow observing Flow Control updates §3.1). Secondly, we adapt the aioquic client [1] to automatically vary configuration parameters in test runs against QUIC servers. We do not run these servers ourselves, but instead make use of the fact that most implementers already provide public Internet endpoints for manual interoperability tests. This allows us to test even non-open source servers and lets us compare configurations between different endpoints backed by the same implementation. For example, the mvfst stack is deployed on a test server and also on Facebook.com, and both setups show marked differences. To eliminate behavioural artefacts due to real network variations, we run our tests a minimum of 5 times on two different WAN networks: first the Hasselt University network (1 Gbps downlink/10Mbps up) and second a residential Wi-Fi network (35Mbps/2Mbps).

We were unable to test all targeted features across all QUIC implementations. Firstly, 3/18 stacks were not considered, as they are not open source, do not provide an endpoint, and/or are not mature enough. For the others, we focus on the 10 most feature-complete and open source stacks (see Table 1). The remaining 5 were tested to the extent possible. In §3 we indicate the amount of stacks evaluated for each feature. After both automated and manual analysis we further validate our results. Firstly, by performing source code reviews where possible. Secondly, by asking each stack's main implementers to confirm and comment on our conclusions, via the quicdev Slack group [4]. As such, almost none of the results presented in this paper are based on conjecture, as most have been explicitly validated by their original developers.

Our results were gathered intermittently over a 4-month period (Jan-Apr 2020) and on IETF QUIC draft versions 25-27. As several implementations considerably changed their behaviours over time (partially due to insights from our work), we re-tested all changed stacks. The results presented here reflect the state of the art in early May 2020. Source code for all our tools, full result analysis sheets, source qlog files and other artefacts can be found at https://qlog.edm.uhasselt.be/epiq.

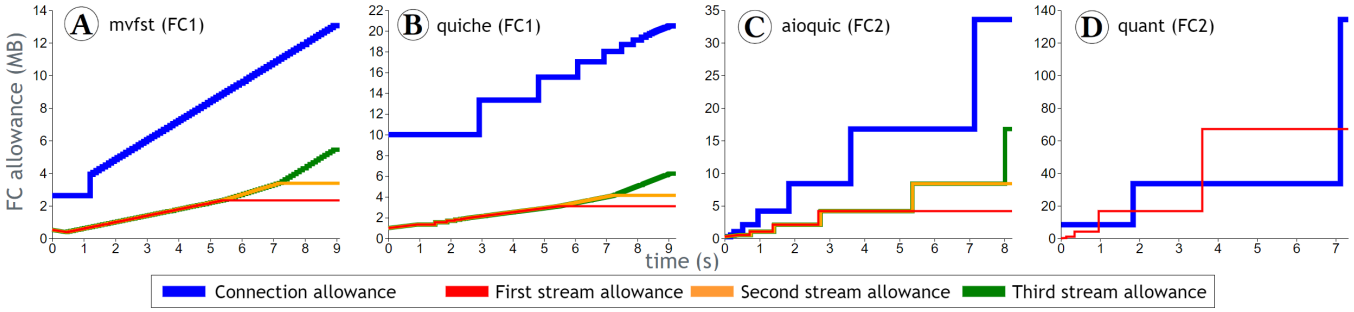

Figure 1: Connection and Stream-level Flow Control allowances for 4 QUIC stacks. A, B and C show concurrent downloads of 3 files (2MB, 3MB, 5MB). D shows a single 10MB download.

3 RESULTS

3.1 Flow Control

When downloading, an endpoint must reserve a transport-level receive buffer to store incoming data, both because data can arrive out-of-order (but can only be delivered to the application-layer in-order) and because the speed at which the application reads from the transport can be lower than the network bandwidth. To prevent overshooting this buffer's capacity, endpoints utilize a Flow Control (FC) system to have the sender match its transmission rate to the speed at which the receive buffer can be emptied. For TCP, which abstracts transported data as a single, ordered byte stream, its singular "receive window" bounds the active bytes in flight allowance and grows and shrinks over time (e.g., a receive window of 0 means a sender should stop sending). In contrast, QUIC allows multiple concurrently active data streams (§3.2), and thus also defines a per-stream FC allowance, in addition to a connection-wide limit. QUIC's limits are expressed in maximum byte stream offsets [24], meaning they can never shrink and only grow in absolute values. Updates to these limits are communicated in MAX(_STREAM)_DATA frames, yet it is up to the developer to decide on the frequency of and allowance amount included in these frames. This is an important aspect to get right, as too few or too low limit updates can stall a fast sender, even if the receive buffer is not fully occupied as the receiver's updates take a RTT to reach the sender [24]. We identify three main approaches (see also Figure 1):

FC1 static allowance Ⓐ: the receive buffer size stays unchanged and the maximum allowance increases linearly.

FC2 growing allowance Ⓒ: the receive buffer size grows over time, causing a non-linear relationship.

FC3 autotuning: the receive buffer size is dynamic, based on RTT estimates and application data consumption rate [40]. Interestingly, we find no current QUIC stacks do the more advanced FC3. Just 3/11 do FC2, while most of the 8/11 stacks doing FC1 simply simply update their absolute FC limits by adding the static buffer size once the receiving application has consumed 50% of the incoming data. Some however do show interesting variations. Firstly, quiche employs FC2 and updates at the 50% mark, but does not add the full buffer size. Instead it adds the amount of bytes the application has consumed (i.e., with every update, the allowance increase is halved, which in turn leads to an increasing update frequency to keep the total allowance static Ⓑ). Secondly, quant uses FC1, but allows stream-level allowance to grow beyond the connection-level limit Ⓓ, which could stall fast senders.

By the implementers' own admission, the presence of these weird behaviours and absence of smarter schemes, is because most have not yet spent time fine-tuning FC approaches and memory requirements. To prevent stalling the sender, many simply set high initial allowances (e.g., 10MB in Ⓑ, 15MB in Google Chrome) and update early (the 50% mark). Many even asked us for guidance in choosing a better FC approach. However, as QUIC is fundamentally different from TCP in this respect, it is difficult to assess which approach works best in practice. Facebook's approach (Ⓐ) does give us an indication, as they have tweaked their behaviour in a real-life deployment. However their setup is also not foolproof (see §3.2), they are biased towards their specific use-case (loading content in native apps), and indicate being limited by existing application layer logic that was originally tweaked for TCP+H2 (e.g., setting higher initial FC limits would cause the app to aggressively preload resources, causing bandwidth contention). In all, we can say QUIC FC is an open problem.

3.2 Multiplexing & Prioritization

TCP abstracts its connection as a single, fully ordered and reliable byte stream. This does perform optimally in situations where multiple, independent data streams can be in progress at the same time (e.g., loading a Web page's resources). H2 attempted to get around this by defining the concept of concurrent byte streams at the application layer, yet this still mapped badly to TCP's single stream viewpoint (e.g., the Head-of-Line (HOL) Blocking problem [27]). This is one of the motivations behind QUIC, which instead makes streams first-class citizens

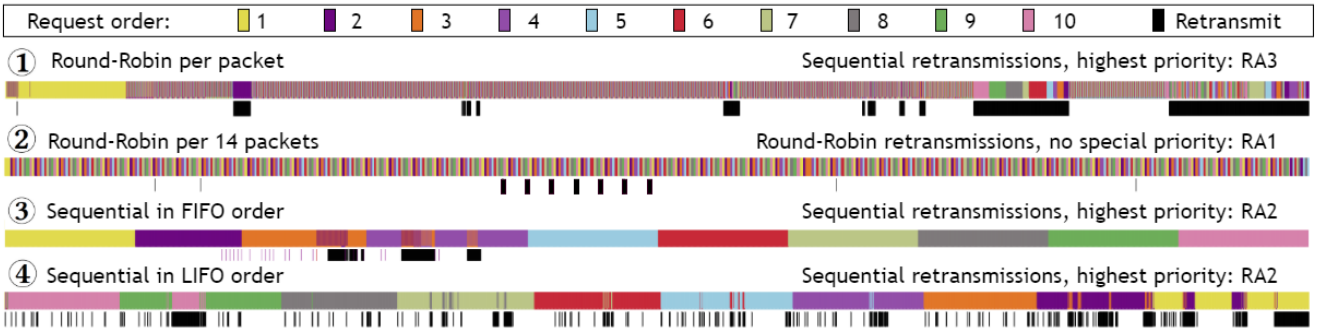

Figure 2: Multiplexing behaviour across different QUIC stacks when downloading 10 1MB files in parallel. Each small colored rectangle is one packet belonging to a file. Long colored areas indicate sequential scheduling. Black areas indicate which frames above them contain retransmitted data. Data arrives from left to right.

in the transport layer. A crucial aspect of handling multiple concurrently active byte streams is how to divide a sender's available bandwidth among them. This can be done in two main ways, see Figure 2. Firstly, a Round-Robin (RR) scheduler (①,②) divides bandwidth among various streams (either fairly or with different stream weights) by splitting resources into smaller chunks and interleaving them. Secondly, a sequential scheduler (③,④) sends all (available) data for a single stream before allocating bandwidth to the next. The optimal approach often depends on the application semantics. For example, both H2 and H3 use a "prioritization" system to drive this behaviour [27, 32, 41] (though H3's system is not yet mature enough to evaluate). However, as QUIC is supposed to be a general purpose transport protocol, it should also have sensible default transport-layer multiplexing. Again though, the QUIC texts leave it fully up to the developer to determine what this behaviour should be. This is clearly visible in the default approaches taken by the different stacks. We find 9/13 stacks to employ a form of RR (6/9 switching streams each QUIC packet, 2/9 switching every 4-10 packets, and 1 unexpectedly switching only after filling the current cwnd (§3.4)). 4/13 stacks opt for a sequential variant instead, though we originally found 3 of them to erroneously sending data in Last-In First-Out (LIFO) order ④, typically thought to be a worst-case approach [38] (2/3 have since changed their approach to FIFO). The optimal approach is however more difficult to determine at this time and requires further study.

A peculiar interaction between Flow Control (FC, §3.1) and stream scheduling was observed when downloading 10 concurrent 1000000 byte (1MB) files from the mvfst server ①. There, a clearly anomalous sequential period is visible for the yellow (first) stream. This was due to our aioquic testclient setting both the connection and stream-level initial FC limits to 1048576 bytes (1MiB). mvfst processes requests 1-by-1 and fully buffered the first file. For the second request, only 48576 bytes remained of the connection FC limit, so only that much was prepared, while requests 3-10 were placed on-hold. When the RR scheduler kicked in, stream 1 was initially multiplexed with stream 2. Soon the stream 2 data ran out and the scheduler had only stream 1 data available, until the client's connection FC update allowed the server to buffer more stream data. It is clear from this example that FC can have (unintended) impactful interactions with stream multiplexing.

A final aspect is how QUIC arranges retransmissions. As TCP's single byte stream abstraction is fully ordered, its retransmissions are always given the absolute highest precedence. However, QUIC's per-stream loss detection and ordering means that retransmissions can be scheduled much like "new" data. Conceptually, we can define 4 Retransmission Approaches (RAs), see also Figure 2. The following example sequences assume a fair RR multiplexer that needs to schedule 8 packets, 2 for each stream A, B, C, and D, where A and B's packets contain retransmissions, versus C and D's new data:

RA1: retransmissions are seen as "normal" data and sent when the scheduler next selects the stream: ABCDABCD.

RA2: retransmissions are given highest precedence, and use the default RR scheduling approach: ABABCDCD.

RA3: retransmissions are given highest precedence, and use a non-default sequential scheduling approach: AABBCDCD.

RA4: retransmissions explicitly take into account application-layer prioritization (e.g., new data for a high priority H3 stream (C) could get precedence over retransmissions of lower priority H3 streams (A and B, with lowest priority D)): CCABABDD. Here we find that most implementers do give retransmissions a higher priority: 10/13 do RA2 and 1/13 (mvfst ①) does RA3, while just 2/13 employ RA1. We have not yet observed RA4 in the wild, though this is likely because only 3/15 stacks integrate QUIC stream scheduling with H3 semantics at this point. It is unclear which RA performs best in practice.

3.3 Packetization

While the binary QUIC and H3 frame and packet structures are well-defined in the specifications, there are many variations in how they can be utilized, sized and combined. For example,

H3 defines the HEADERS and DATA frames [7]. These are in turn passed to the QUIC layer, whereby typically QUIC has no knowledge of the H3 semantics: it treats H3-level data on each QUIC stream as an ordered but opaque byte sequence. These bytes are then put inside QUIC-level STREAM frames for transport. Practically, this means that multiple H3-level frames can be aggregated together inside a single QUIC STREAM frame, which is good for efficiency. We find that 9/13 servers consistently do this, but that 4/13 tend to instead pack HEADERS and DATA frames into separate STREAM frames. In stress tests that request hundreds of very small files (<1kB), 6/13 servers started showing even more inefficient behaviour, packing all H3 frames in separate STREAM frames, and even in tiny QUIC packets. If we define goodput efficiency as the amount of useful transported H3-level data (e.g., image file bytes) divided by the total amount of bytes on the wire (including QUIC and H3 framing overhead), we find that most stacks achieve 95-97% efficiency when downloading larger files, which plummets to about 90% for most when downloading many smaller files, with the worst case only achieving 83%.

A part of this is the sizing of H3 DATA frames. While QUIC STREAM frames cannot span multiple QUIC packets, H3 DATA frames can theoretically be up to 4600 Petabytes (as their length is 64-bit encoded), and thus span many STREAM frames. For goodput efficiency, fewer and thus larger DATA frames are best. Here, we find a large heterogeneity. When downloading files larger than 1MB, 6/13 have DATA frames larger than 1MB, 2/13 between 1MB and 100kB, and 5/13 lower than 100kB (of which one has the worst case of generating a new DATA frame for each QUIC packet). Interestingly, we observed two stacks that dynamically sized their DATA frames, growing larger over time, tied as they are to the current QUIC Flow Control and Congestion Window values (§3.1, §3.4). While this seemed like intentional behaviour, it again turned out to be due to unexpected cross-layer code interactions.

Finally, QUIC mandates a minimum UDP payload size of 1200 bytes [24], but it is generally understood that larger sizes significantly improve efficiency [20, 25]. It is best practice to start with a small packet size and perform Path MTU Discovery (PMTUD) [24]. Still, we find that at this time, just 3/14 stacks implement PMTUD, all of them using the naive method of sending a single 1400-1500 byte QUIC packet containing mainly PADDING instead of the more advanced DPLPMTUD approach [15]. The need for PMTUD was emphasized to us by Facebook, who find many networks with higher loss rates if QUIC packets are even a few dozen bytes larger.

3.4 Congestion Control

In terms of recovery (loss detection and congestion control (CC)), QUIC inherits most of TCP's concepts and decades of best practices (e.g., selective acknowledgements, pacing, tail loss probes). The QUIC recovery text [22] aggregates a discussion of all these concepts with how they can be practically adapted to QUIC peculiarities such as its integrated TLS handshake. For a practical example with pseudo-code, the somewhat outdated, yet well-understood New Reno CC [22] is used. As such, the text provides a good starting point for adapting other CCs to QUIC. We find that while most of the stacks have implemented QUIC's New Reno variant (9/15), especially many of the larger companies indeed also support more modern CCs: 6/15 implement Cubic (4 with hystart [12], 1 with tweaks for satellite networks [21]), 4/15 implement BBRv1 and 3/15 go further with approaches like COPA [6] or BBRv2 [10]. Facebook deploys BBRv1, Cloudflare Cubic [12].

An important CC variable is the initial Congestion Window (cwnd), which controls how many bytes an endpoint can send back in the first flight (before growing the cwnd in "slow start"). The QUIC text's advice of an initial cwnd of 13kB-15kB is followed by 10/15 stacks, while 3/15 choose a much larger window of 40kB+, though these values are more heterogeneous in actual deployments [36]. For example, we learned that Facebook uses machine learning to tune their init cwnd, while f5 includes a cwnd estimate in their address validation token for resumed connections (§3.5).

Additionally, the QUIC text strongly encourages the use of pacing (i.e., spreading out packets over an entire RTT instead of sending them in a single burst with each cwnd increase, which is thought to lower packet loss [5]). Interestingly, only 8/15 currently support this. This is mainly due to the complexity of the technique and lacking support in the Linux kernel in combination with other optimization techniques (e.g., GSO combined with SO_TXTIME [14, 20, 25]).

Finally, the performance of a CC can be influenced heavily by the frequency with which the receiver acknowledges (ACKs) data. QUIC recommends sending an ACK for every 2 received packets [22]. Just 3/12 follow this recommendation, with 2/12 ACKing every packet instead, and 7/10 ACKing every 2-10 packets. This latter behaviour is mostly due to implementations reading up to 10 packets at a time from the socket. It is however also understood that ACK processing is expensive in QUIC [19, 25] and 4/15 are experimenting with the ACK Frequency extension to reduce overhead [23]

3.5 0-RTT

One of the key new features in QUIC is the zero RTT (0RTT) connection setup [17], which allows the exchange of application data (e.g., an H3 GET and its (partial) response) in the first flight (compared to third or fourth in TCP+TLS). This derives from TLS 1.3, which allows exchanging Pre-Shared encryption Keys in Session Tickets during a first "1RTT" connection (where data can only be exchanged from the second flight onward), which is then used to enable 0RTT on a subsequent

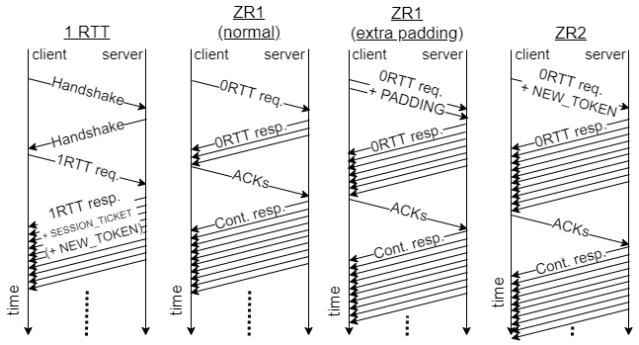

Figure 3: 0-RTT request and response size variations. We assume an initial cwnd of 10 packets at the server.

connection [39], see Figure 3. Despite being a high-profile feature, just 13/18 implement it, of which we tested 9.

One of the reasons for this lower uptake is that 0RTT is complex to implement securely, as it is vulnerable to (HTTP) replay and UDP amplification/reflection attacks [24, 39]. This latter category is possible when the attacker spoofs their IP address and sends a (small) 0RTT request for a (large) resource to the server. If the server simply starts sending the (entire) resource to the spoofed victim IP, it could be used in a (D)DoS attack. To prevent this, a QUIC server MUST NOT send more than 3 times as much data as it has received from the client until the path is validated (confirming the IP was not spoofed). This validation can happen in 3 main ways (ZRs) (Figure 3):

ZR1 waits for a reply from the client to the early 0RTT server packets. This has the large downside that the 0RTT response will be rather small (just 5kB-7kB if the client sends its initial request in 1-2 packets). We feel this significantly reduces 0RTT's usefulness for typical web browsing use cases.

ZR2 alleviates this by sending an Address Validation token in QUIC's NEW_TOKEN frame [24]. This is sent encrypted by the server in the first connection and used by the client for the second, so the server can consider the path validated immediately. This allows it to ignore the 3X limit and send more data, typically up to its initial cwnd (10-40kB §3.4).

ZR3 is mainly a legacy equivalent of ZR2 which securely encodes the client's IP address inside the TLS Session Ticket. ZR1 and ZR2 are both used in 6/13 stacks, but ZR3 only by Facebook, who have plans to migrate to the superior ZR2.

One way to improve upon ZR1 would be for the client to send additional data along with the 0RTT request (e.g., in the form of padding), see Figure 3. This would in turn allow the server to reply with more data while still adhering to the 3X limit. While testing whether the stacks would respond well to this, we found several high impact bugs. One stack simply did not adhere to the 3X limit, replying up to their 46kB initial cwnd to a 1.2kB request (a 36X amplification). Another did apply the 3X limit, but forgot to check its initial cwnd for 0RTT responses (e.g., replying with 30kB 0RTT data to a 10kB request even though their cwnd was only 15kB). Finally, one stack forgot to account retransmissions of lost packets in its 3X limit. If an attacking client never replied to anything after its first 1.2kB, this stack sent up to 17kB of (retransmitted) data (14X). Most other servers did adhere to the 3X limit and also sent more data in response to a client sending additional padding. As such, we recommend clients pad their 0RTT requests to about 4kB-5kB (higher values will give diminishing returns as most servers utilize an initial cwnd of about 13kB (§3.4)). This should not be needed for servers employing ZR2 (which should be preferred over ZR3).

4 DISCUSSION & CONCLUSION

In this work, we have discussed 15 different QUIC implementations across a multitude of behaviours (see Table 1). Even though these stacks all implement the exact same QUIC/H3 protocols, we have shown that their low-level implementation choices lead to a large behavioural heterogeneity between them. We believe this has important consequences for the ways in which QUIC/H3 can and should be evaluated.

While not all considered aspects might have a large impact on most types of protocol evaluation results (e.g., H3 DATA frame sizing or PMTUD support for Web performance research [9]), other discussed features, such as Flow Control, Congestion Control, Prioritization and 0 RTT can all lead to significant differences in results. Yet, these are aspects that historically we rarely see evaluated or discussed in related work focusing on for example H2 [18, 41] and also gQUIC [16]. In order to be able to draw solid conclusions about QUIC/H3 as protocols, we feel that researchers should show scientific rigor in two main ways. Firstly, by performing deep root-cause analysis of all observed high-level behaviours. Secondly, by comparing multiple QUIC implementations. This especially holds true in the next few years (2020-2023), as not all implementations will be fully optimized or complete by the time QUIC/H3 are finalized. Even later on, we suspect stacks will remain heterogeneous and deep insight will remain key in researching and optimizing QUIC and H3.

We believe that our methodology of using the qlog and qvis tools [26, 29] has proven its potential to form the basis of a framework to both analyze and extend or improve QUIC/H3 stacks. This is also evidenced by the fact that several QUIC implementers have lately started using these tools to validate their approaches [12]. As currently 12/18 QUIC stacks support qlog, these tools are broadly available and ready to use.

Overall, we posit that QUIC stacks are becoming mature enough to be deployed and researched, but results from high-level metrics should be thoroughly root-cause analyzed if researchers want to draw broad conclusions on QUIC/H3 as protocols. There are many opportunities for future research on QUIC behaviour tuning, especially around Flow Control, Multiplexing/H3 Prioritization, and Retransmission approaches.

REFERENCES

[1] 2020. (2020). https://github.com/rmarx/aioquic.

[2] 2020. Active QUIC implementations. (2020). https://github.com/quicwg/base-drafts/wiki/Implementations.

[3] 2020. QUIC Interop Runner. (2020). https://interop.seemann.io/.

[4] 2020. QUICdev Slack Group. (2020). https://quicdev.slack.com/.

[5] 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.

[6] Venkat Arun and Hari Balakrishnan. 2018. Copa: Practical delay-based congestion control for the internet.

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

[8] Prasenjeet Biswal and Omprakash Gnawali. 2016. Does quic make the web faster?. In *2016 IEEE GLOBECOM*. IEEE.

[9] Enrico Bocchi, Luca De Cicco, and Dario Rossi. 2016. Measuring the quality of experience of web users. *ACM SIGCOMM Computer Communication Review* 46, 4 (2016).

[10] Neal Cardwell, Yuchung Cheng, C Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. 2016. BBR: Congestion-based congestion control. (2016).

[11] Gaetano Carlucci, Luca De Cicco, and Saverio Mascolo. 2015. HTTP over UDP: an Experimental Investigation of QUIC. In *Proc. of SAC*.

[12] Junho Choi. 2020. CUBIC and HyStart++ Support in quiche. (2020). https://blog.cloudflare.com/cubic-and-hystart-support-in-quiche/.

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

[14] Willem de Bruijn and Eric Dumazet. 2018. Optimizing UDP for content delivery: GSO, pacing and zerocopy. In *Linux Plumbers Conference*.

[15] Fairhurst et al. 2020. *Packetization Layer Path MTU Discovery for Datagram Transports.* Internet-Draft draft-ietf-tsvwg-datagram-plpmtud-20. https://datatracker.ietf.org/doc/html/draft-ietf-tsvwg-datagram-plpmtud-20

[16] Kakhki et al. 2017. Taking a long look at QUIC: an approach for rigorous evaluation of rapidly evolving transport protocols. In *Proc. of IMC*.

[17] Langley et al. 2017. The quic transport protocol: Design and internet-scale deployment. In *Proc. of SIGCOMM*.

[18] Wolsing et al. 2019. A performance perspective on web optimized protocol stacks: TCP+ TLS+ HTTP/2 vs. QUIC. In *Proc. of ANRW*.

[19] Gorry Fairhurst and Ana Custura. 2020. Changing QUIC default to ACK 1:10. (2020). https://erg.abdn.ac.uk/users/gorry/ietf/QUIC/QUIC-ack10-24-april-00.pdf.

[20] Alessandro Ghedini. 2020. Accelerating UDP packet transmission for QUIC. (2020). https://blog.cloudflare.com/accelerating-udp-packet-transmission-for-quic/.

[21] Christian Huitema. 2020. Faster slow start for satellite links? (2020). https://huitema.wordpress.com/2020/04/21/faster-slow-start-for-satellite-links/.

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

[23] Jana Iyengar and Ian Swett. 2020. *Sender Control of Acknowledgement Delays in QUIC.* Internet-Draft draft-iyengar-quic-delayed-ack-00. https://datatracker.ietf.org/doc/html/draft-iyengar-quic-delayed-ack-00

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

[25] Jana Iyengar Kazuho Oku. 2020. Can QUIC match TCP's computational efficiency? (2020). https://www.fastly.com/blog/measuring-quic-vs-tcp-computational-efficiency.

[26] Robin Marx. 2020. qvis toolsuite live. (2020). https://qvis.edm.uhasselt.be.

[27] Robin Marx, Tom De Decker, Peter Quax, and Wim Lamotte. 2019. Of the Utmost Importance: Resource Prioritization in HTTP/3 over QUIC.

[28] Robin Marx, Wim Lamotte, Jonas Reynders, Kevin Pittevils, and Peter Quax. 2018. Towards QUIC debuggability. In *Proc. of EPIQ workshop*.

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

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

[31] Péter Megyesi, Zsolt Krämer, and Sándor Molnár. 2016. How quick is QUIC?. In *2016 IEEE ICC*. IEEE.

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

[33] Mirko Palmer, Thorben Krüger, Balakrishnan Chandrasekaran, and Anja Feldmann. 2018. The quic fix for optimal video streaming. In *Proc. of EPIQ Workshop*.

[34] James Pavur, Martin Strohmeier, Vincent Lenders, and Ivan Martinovic. 2020. QPEP: A QUIC-Based Approach to Encrypted Performance Enhancing Proxies for High-Latency Satellite Broadband. (2020).

[35] Maxime Piraux, Quentin De Coninck, and Olivier Bonaventure. 2018. Observing the evolution of QUIC implementations. In *Proc. of EPIQ workshop*.

[36] Jan Rüth, Ike Kunze, and Oliver Hohlfeld. 2019. TCP's Initial Window—Deployment in the Wild and Its Impact on Performance. *IEEE Transactions on Network and Service Management* 16, 2 (2019).

[37] Darius Saif, Chung-Horng Lung, and Ashraf Matrawy. 2020. An Early Benchmark of Quality of Experience Between HTTP/2 and HTTP/3 using Lighthouse. (2020).

[38] Ian Swett and Robin Marx. 2019. HTTP Priority design team update - IETF 107. (18 November 2019). https://github.com/httpwg/wg-materials/blob/gh-pages/ietf106/priorities.pdf.

[39] Martin Thomson and Sean Turner. 2020. *Using TLS to Secure QUIC.* Internet-Draft draft-ietf-quic-tls-27. https://datatracker.ietf.org/doc/html/draft-ietf-quic-tls-27

[40] Eric Weigle and Wu-chun Feng. 2002. A comparison of TCP automatic tuning techniques for distributed computing. In *Proc. of HPDC Symposium*. IEEE.

[41] Maarten Wijnants, Robin Marx, Peter Quax, and Wim Lamotte. 2018. Http/2 prioritization and its impact on web performance. In *Proc. of WWW*.