Web Performance Calendar

The speed geek's favorite time of year
2018 Edition

QUIC and HTTP/3 : Too big to fail?!

4thDec 2018 by Robin Marx
ABOUT THE AUTHOR



Robin Marx is a Web Performance PhD candidate at Hasselt University, Belgium. He is mainly looking into HTTP/2 and QUIC performance, and maintains the TypeScript QUIC implementation Quicker. In a previous life he was a multiplayer game programmer and co-founder of LuGus Studios. YouTube videos of Robin are either humoristic technical talks or him hitting other people with longswords.

QUIC and HTTP/3 : Too big to fail?!

The new QUIC and HTTP/3 protocols are coming and they are the bee's knees! Combining lessons and best practices from over 30 years of networking, the new protocol stack offers major improvements to performance, privacy, security and flexibility.

Much has been said about the potential benefits of QUIC, most of it based on Google's experience with an early version of the protocol. However, its potential shortcomings are rarely talked about and little is yet known about the properties of the upcoming, standardized versions (as they are still under active development). This post takes a (nuanced) "devil's advocate" viewpoint and looks at how QUIC and HTTP/3 might still fail in practice, despite the large amount of current enthusiasm. In all fairness, I will also mention counter arguments to each point and let the reader make up their own mind, hopefully after plenty of additional discussion.

> Note: if you're not really sure what QUIC and HTTP/3 are in the first place, it's best to get up to speed a bit before reading this post, which assumes some familiarity with the topic. Some resources that might help you with that:

- Mattias Geniar's blog post
- Cloudflare's write-up
- Robert Graham's comments
- Daniel Stenberg (@bagder)'s HTTP/3 explained
- Mailing list explanation and blog post by Patrick McManus
- And my own talk from DeltaVConf this year

1. End-to-end encrypted UDP you say?

One of the big selling points of QUIC is its end-to-end encryption. Where in TCP much of the transport-specific information is out in the open and only the data is encrypted, QUIC encrypts almost everything and applies integrity protection (see Figure **X**). This leads to improved privacy and security and prevents middleboxes in the network from tampering with the protocol. This last aspect is one of the main reasons for the move to UDP: evolving TCP was too difficult in practice because of all the disparate implementations and parsers.

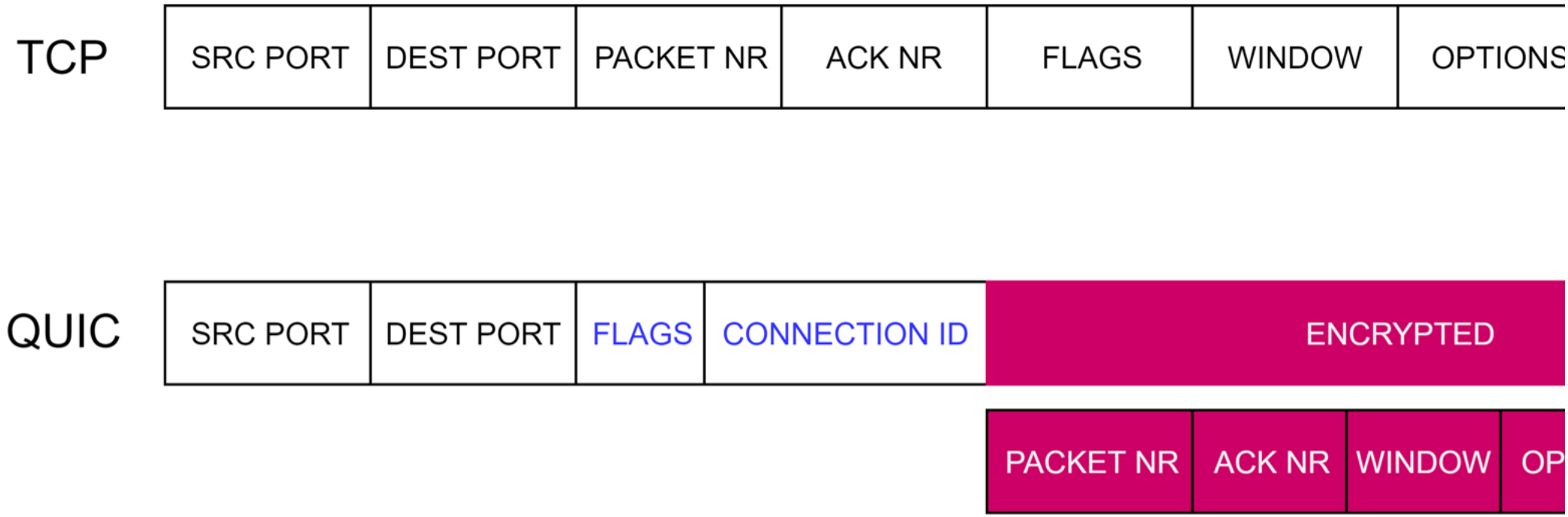


Figure X: simplified conceptual representation of the (encrypted) fields in TCP and QUIC

Network operators and the spin bit

The downside is that network operators now have much less to work with to try and optimize and manage their network. They no longer know if a packet is an acknowledgment or a re-transmit, cannot self-terminate a connection and have no other way from impacting congestion control/send rate than to drop packets. It is also more difficult to assess for example the round-trip-time (RTT) of a given connection (which, if rising, is often a sign of congestion or bufferbloat).

There has been much discussion about adding some of these signals back into a visible-on-the-wire part of the QUIC header (or using other means), but the end result is that just a single bit will be exposed for RTT measurement: the "spin" bit. The concept is that this bit will change value about once every round trip, allowing middleboxes to watch for the changes and estimate RTTs that way, see Figure **Y** (more bits could lead to added resolution etc., read this excellent paper). While this helps a bit, it still limits the operators considerably, especially with initial signals being that Chrome and Firefox will not support the spin bit. The only other option QUIC will support is "Explicit Congestion Notification", which uses flags at the IP-level to signal congestion.

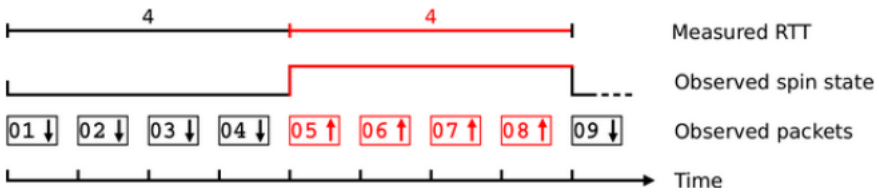


Figure Y: A simple illustration of the working of the spin bit (source)

UDP blocking and alt-svc with fallbacks

I don't know about you, but if I were a network operator (or nefarious dictatorship), I would be sorely tempted to just block QUIC wholesale if I'm doing any type of TCP optimization or use special security measures. It wouldn't even be that difficult for web-browsing: nothing else runs on UDP:443 (whereas blocking TCP:443 would lead to much mayhem). While deploying QUIC, google actually looked at this, to know how many networks just blocked UDP/QUIC already. They (and other research) found 3-5% of networks currently do not allow QUIC to pass. That seems fine, but these figures (probably) don't include a lot of corporate networks, and the real question is: will it remain that way? If QUIC gets bigger, will (some) networks not start actively blocking it (at least until they update their firewalls and other tools to better deal with it?). "Fun" anecdote: while testing our own QUIC implementation's public server (based in Belgium) with the excellent quic-tracker conformance testing tool, most of the tests suddenly started failing when the tool moved to a server in Canada. Further testing confirmed some IP-paths are actively blocking QUIC traffic, causing the test failures.

The thing is that blocking QUIC (e.g., in a company's firewall) wouldn't even break anything for web-browsing end users; sites will still load. As browsers (and servers!) have to deal with blocked UDP anyway, they will always include a TCP-based fallback (in practice, Chrome currently even races TCP and QUIC connections instead of waiting for a QUIC timeout). Servers will use the alt-svc mechanism to signal QUIC support, but browsers can only trust that to a certain extent because a change of network might suddenly mean QUIC becomes blocked. QUIC-blocking company network administrators won't get angry phone calls from their users and will still be able to have good control over their setup, what's not to like? They also won't need to run and maintain a separate QUIC/H3 stack next to their existing HTTP(/2) setup..

Finally, one might ask: why would a big player such as Google then want to deploy QUIC on their network if they loose flexibility? In my assessment, Google (and other large players) are mostly in full control of (most of) their network, from servers to links to edge points-of-presence, and have contracts in place with other network operators. They know more or less exactly what's going on and can mitigate network problems by tweaking load balancers, routes or servers themselves. They can also do other shenanigans, such as encode information in one of the few non-encrypted fields in QUIC: the connection-ID. This field was explicitly allowed to be up to 18 bytes long to allow encoding (load-balancing) information inside. They could also conceivably add additional headers to their packets, stripping them off as soon as traffic leaves the corporate network. As such, the big players lose a bit, but not much. The smaller players or operators of either only servers or only the intermediate networks stand to lose more.

Counterarguments

- End-users will clamor for QUIC to be allowed because of the (performance) benefits it provides
- QUIC doesn't need performance enhancing middleboxes anyway because it has better built-in congestion control and faster connection setup
- Most current networks don't block it, little chance they will start without a major reason/incident
- Running an QUIC+HTTP/3 stack next to TCP+HTTP/2 will be as easy as adding a couple of lines to a server config

2. CPU issues

As of yet, QUIC is fully implemented in user-space (as opposed to TCP, which typically lives in kernel-space). This allows fast and easy experimentation, as users don't need to upgrade their kernels with each version, but also introduces severe performance overheads (mainly due to user-to-kernel-space communication) and potential security issues.

In their seminal paper, Google mentions their server-side QUIC implementation uses about **2x** as much CPU as the equivalent TCP+TLS stack. This is already after some optimizations, but not full kernel bypass (e.g., with DPDK or netmap). Let me put that another way: they would need roughly **twice the server hardware** to serve the same amount of traffic! They also mention diminished performance on mobile devices, but don't give numbers. Luckily, another paper describes similar mobile tests and finds that Google's QUIC is mostly still faster than TCP but "QUIC's advantages diminish across the board", see Figure **Z**. This is mainly because QUIC's congestion control is "application limited" **58% of the time** (vs 7% on the desktop), meaning the CPU simply cannot cope with the large amount of incoming packets.

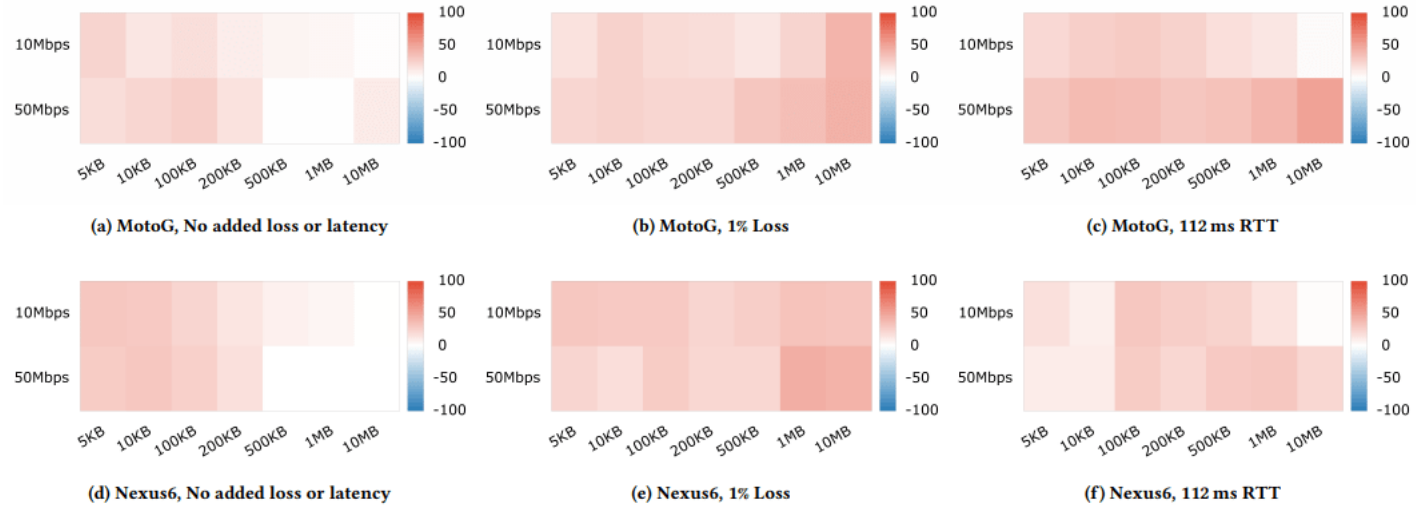


Figure 12: QUICv34 vs. TCP for varying object sizes on MotoG and Nexus6 smartphones (using WiFi). We find that QUIC's improvements diminish or disappear entirely when running on mobile devices.

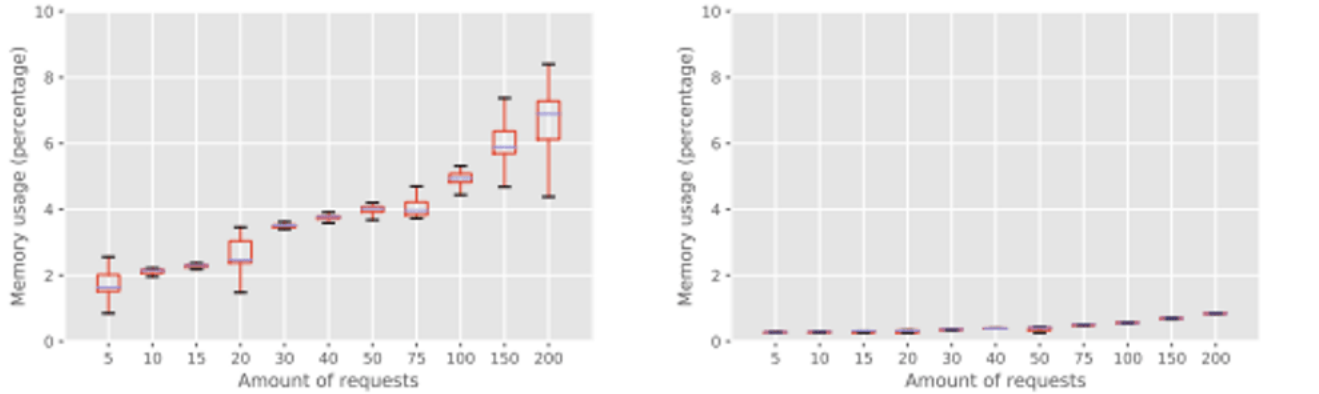
Figure Z: QUIC vs TCP performance. Red = QUIC better, Blue = TCP better. (source)

This would suggest QUIC provides most advantages over TCP in situations with bad networks and high-end devices. Sadly, bad networks are often coupled with bad devices, and the median global network and device are both quite slow. This means a lot of the network gains from QUIC are potentially (largely) undone by the slower hardware. Combine this with the fact that webpages themselves are also asking more and more CPU for themselves (leading to one web performance guru claiming JavaScript perf is more important than the network nowadays), and you've got quite the conundrum.

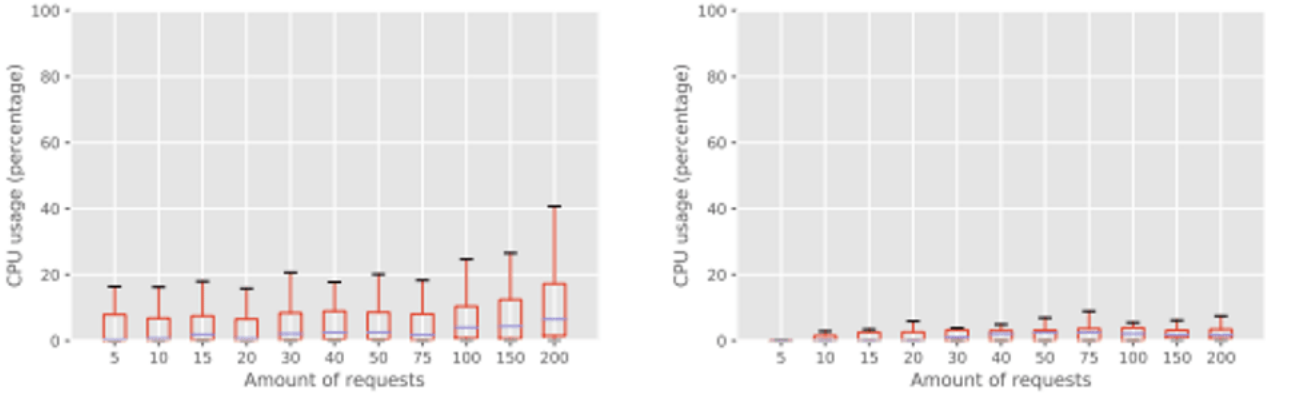
IoT and TypeScript

One of the oft-touted use cases for QUIC is in Internet-of-Things (IoT) devices, as they often need intermittent (cellular) network access and low-latency connection setup, 0-RTT and better loss resilience are quite interesting in those cases. However, those devices often also have quite slow CPUs.. There are many issues where QUIC's designers mention the IoT use case and how a certain decision might impact this, though as far as I know there is no stack that has been tested on such hardware yet. Similarly, many issues mention taking into account a hardware QUIC implementation, but at my experience level it's unclear if this is more wishful thinking and handwaving rather than a guarantee.

I am a co-author of a NodeJS QUIC implementation in TypeScript, called Quicker. This seems weird given the above, and indeed, most other stacks are in C/C++, Rust or Go. We chose TypeScript specifically to help assess the overhead and feasability of QUIC in a scripting language and, while it's still very early, it's not looking too well for now, see Figure **A**.



(a) Memory usage over various request counts of Quicker. (b) Memory usage over various request counts of ngtcp2.



(a) CPU usage over various request counts of Quicker. (b) CPU usage over various request counts of ngtcp2.

Figure A: Quicker (TypeScript) vs ngtcp2 (C/C++) CPU and memory usage (source)

Counterarguments

- QUIC will move into kernel and/or hardware in the future
- TCP+TLS overhead account for almost nothing when compared to other overheads (e.g., PHP execution, database access). QUIC taking twice that is negligible.
- Current numbers are for google's QUIC, IETF QUIC can/will be different
- (Client) hardware will become faster
- The overhead is not that high as to be unmanageable
- Even with a massive overhead, Google decided to deploy QUIC at scale. This indicates the benefits (severely) outweigh the costs. It would seem better web performance indeed leads to massively improved revenues, who knew?
- TCP also has a place in IoT
- "I've looked at your TypeScript code Robin, and it's an ungodly mess. A competent developer could make this way faster"

3. 0-RTT usefulness in practice

Another major QUIC marketing feature (though it's actually from TLS 1.3) is 0-RTT connection setup: your initial (HTTP) request can be bundled with the first packet of the handshake and you can get data back with the first reply, superfast!

However, there is a "but" immediately: this only works with a server that we've previously connected to with a normal, 1-RTT setup. 0-RTT data in the second connection is encrypted with something called a "pre-shared secret" (contained in a "new session ticket"), which you obtain from the first connection. The server also needs to know this secret, so you can only 0-RTT connect to that same server, not say, a server in the same cluster (unless you start sharing secrets or tickets etc.). This means, again, that load balancers should be smart in routing requests to correct servers. In their original QUIC deployment, Google got this working in 87% (desktop) – 67% (mobile) of resumed connections, which is quite impressive, especially since they also required users to keep their original IP addresses.

There are other downsides as well: 0-RTT data can suffer from "replay attacks", where the attacker copies the initial packet and sends it again (several times). Due to integrity protection, the contents cannot be changed, but depending on what the application-level request carries, this can lead to unwanted behaviour if the request is processed multiple times (e.g., POST bank.com?addToAccount=1000). Thus, only what they call "idempotent" data can be sent in 0-RTT (meaning it should not permanently change state, e.g., HTTP REST GET but not PUT). Depending on the application, this can severely limit the usefulness of 0-RTT (e.g., a naive IoT sensor using 0-RTT to POST sensor data could, conceptually, be a bad idea).

Lastly, there is the problem of IP address spoofing and the following UDP amplification attacks. In this case, the attacker pretends to be the victim at IP a.b.c.d and sends a (small) UDP packet to the server. If the server replies with a (much) larger UDP packet to a.b.c.d, the attacker needs much less bandwidth to generate a large attack on the victim, see Figure **B**. To prevent this, QUIC adds two mitigations: the client's first packet needs to be at least 1200 bytes (max practical segment size is about 1460) and the server MUST NOT send more than three times that in response without receiving a packet from the client in response (thus "validating the path", proving the client is not a victim of an attack). So just 3600-4380 bytes, in which the TLS handshake and QUIC overhead is also included, leaves little space for an (HTTP) response (if any). Will you send the HTML `<head>`? headers? Push something? Will it matter? This exact question is one of the things I'm looking forward to investigating in-depth.

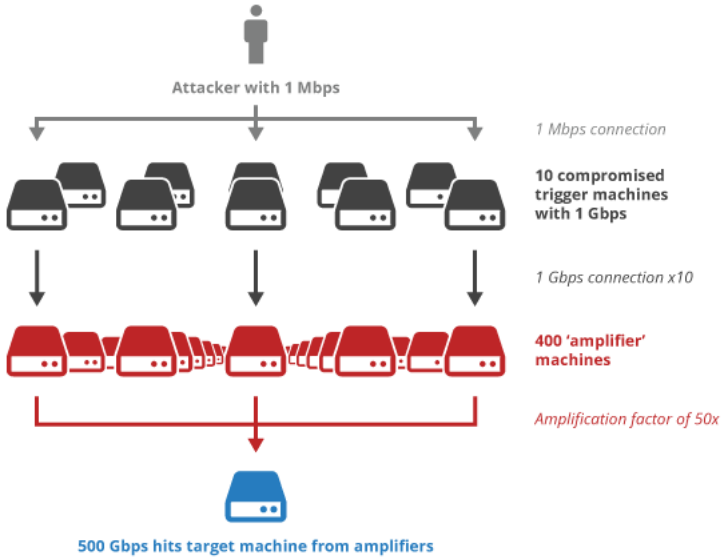


Figure B: UDP amplification attack (source)

The final nail in QUIC's coffin is that TCP + TLS 1.3 (+ HTTP/2) can also use 0-RTT with the TCP "Fast Open" option (albeit with the same downsides). So picking QUIC just for this feature is (almost) a non-argument.

Counterarguments

- HTTP servers should already be immune to replays
- Replay prevention is easy when adding something like a timestamp or sequence number, since attackers cannot change the contents
- 1-RTT is still much better than the current 3-4 RTT with TCP + TLS 1.2
- Servers can just ignore the RFCs artificial UDP amplification limit of 3600 bytes if performance is needed. Bigger players could deploy other measures, like keeping track of how much is sent to a given IP at once, take into account server load or total outgoing bandwidth. Smaller players might not have that luxury though.
- There is some evidence that larger initial congestion windows do not have a big overall impact on HTTP/2 over TCP so the limit might not matter much in practice for H3
- TCP Fast Open is currently unusable in practice. Mozilla even gave up on trying to enable it in Firefox. (Counter-counterargument: over time, TCP Fast Open support will also increase)

4. QUIC v3.5.66.6.8.55-Facebook

As opposed to TCP, QUIC integrates a full version negotiation setup, mainly so it can keep on evolving easily without breaking existing deployments. The client uses its most preferred supported version for its first handshake packet. If the server does not support that version, it sends back a Version Negotiation packet, listing supported versions. The client picks one of those (if possible) and retries the connection. This is needed because the binary encoding of the packet can

change between versions.

Every RTT is one too many

As follows from the above, each version negotiation takes 1 RTT extra. This wouldn't be a problem if we have a limited set of versions, but the idea seems to be that there won't be just for example 1 official version per-year, but a slew of different versions. One of the proposals was (is?) even to use different versions to indicate support for a single feature (the previously mentioned spin bit). Another goal is to have people use a new version when they start experimenting with different, **non-standardized** features. This all can (will?) lead to a wild-wild-west situation, where every party starts running their own slighty different versions of QUIC, which in turn will increase the amount of instances that version negotiation (and the 1 RTT overhead) occurs. Taking this further, we can imagine a dystopia where certain parties refuse to move to new standardized versions, since they consider their own custom versions superior. Finally, there is the case of drop-and-forget scenarios, for example in the Internet-of-Things use case, where updates to software might be few and far between.

A partial solution could potentially be found in the transport parameters. These values are exchanged as part of the handshake and could be used to enable/disable features. For example, there is already a parameter to toggle connection migration support. However, it's not yet clear if implementers will lean to versioning or adding transport parameters in practice (though I read more of the former).

It seems a bit strange to worry about sometimes having a 1-RTT version negotiation cost, but for a protocol that markets a 0-RTT connection setup, it's a bit contradictory. It is not inconceivable that clients/browsers will choose to always attempt the first connection at the lowest supported QUIC version to minimize the 1-RTT overhead risk.

Counterarguments

- Browsers will only support "main" versions and as long as your server supports those you should be ok
- Parties that run their own versions will make sure both clients and servers support those or make the 1-RTT trade-of decision consciously
- Clients will cache the version lists servers support and choose a supported version from the second connection onward
- Versions used for toggling individual features can easily share a single codebase. Servers will be smart enough not to launch negotiation if they don't support that **exact** version, if they know the on-the-wire image is the same and they can safely ignore the missing feature
- Servers always send back their full list of supported versions in their transport parameters, even without version negotation. From the second connection onward, the client can select the highest mutually supported setup.

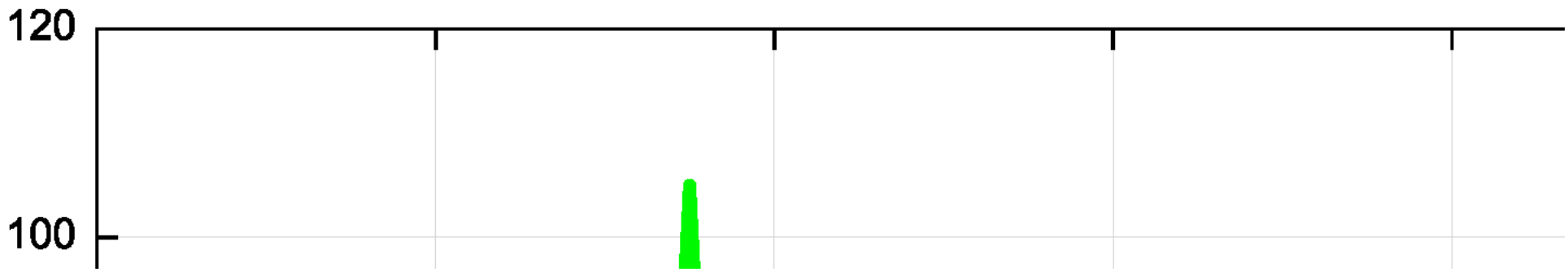
5. Fairness in Congestion Control

The fact that QUIC is end-to-end encrypted, provides versioning and is implemented in user space provides a never-before seen amount of flexibility. This really shines when contemplating using different congestion control algorithms (CCAs). Up until now, CCAs were implemented in the kernel. You could conceivably switch which one you used, but only for your entire server at the same time. As such, most CCAs are quite general-purpose, as they need to deal with any type of incoming connection. With QUIC, you could potentially switch CCA on a per-connection basis (or do CC across connections!) or at least more easily experiment with different (new) CCAs. One of the things I want to look at is using the NetInfo API to get the type of incoming connection, and then change the CCA parameters based on that (e.g., if you're on a gigabit cable, my first flight will be 5MB instead of 14KB, because I know you can take it).

Calimero

The previous example clearly highlights the potential dangers: if anybody can just decide what to do and tweak their implementations (without even having to recompile the kernel- madness!), this opens up many avenues for abuse. After all, an important part of congestion control is making sure each connection gets a more or less equal share of the bandwidth, a principle called **fairness**. If some QUIC servers start deploying a much more aggressive CCA that grabs more than its equal share of bandwidth, this will slow down other, non-QUIC connections and other QUIC connections that use a different CCA.

Nonsense, you say! Nobody would do that, the web is a place of gentlepeople! Well… Google's version of QUIC supports two congestion control algorithms: TCP-based CUBIC and BBR. There is some conflicting information, but at least some sources indicate their CCA implementations are severely unfair to "normal" TCP. One paper, for example, found that QUIC+CUBIC used twice the bandwidth of 4 normal TCP+CUBIC flows **combined**. Another blogpost shows that TCP+BBR could scoop up two-thirds of the available bandwidth, see Figure **C**. This is not to say that Google actively tries to slow down other (competing) flows, but it shows rather well the risks with letting people easily choose and tweak their own CCAs. Worst case, this can lead to an "arms race" where you have to catch up and deploy ever more aggressive algorithms yourself, or see your traffic drowned in a sea of QUIC packets. Yet another potential reason for network operators to block or severely hamper QUIC traffic.



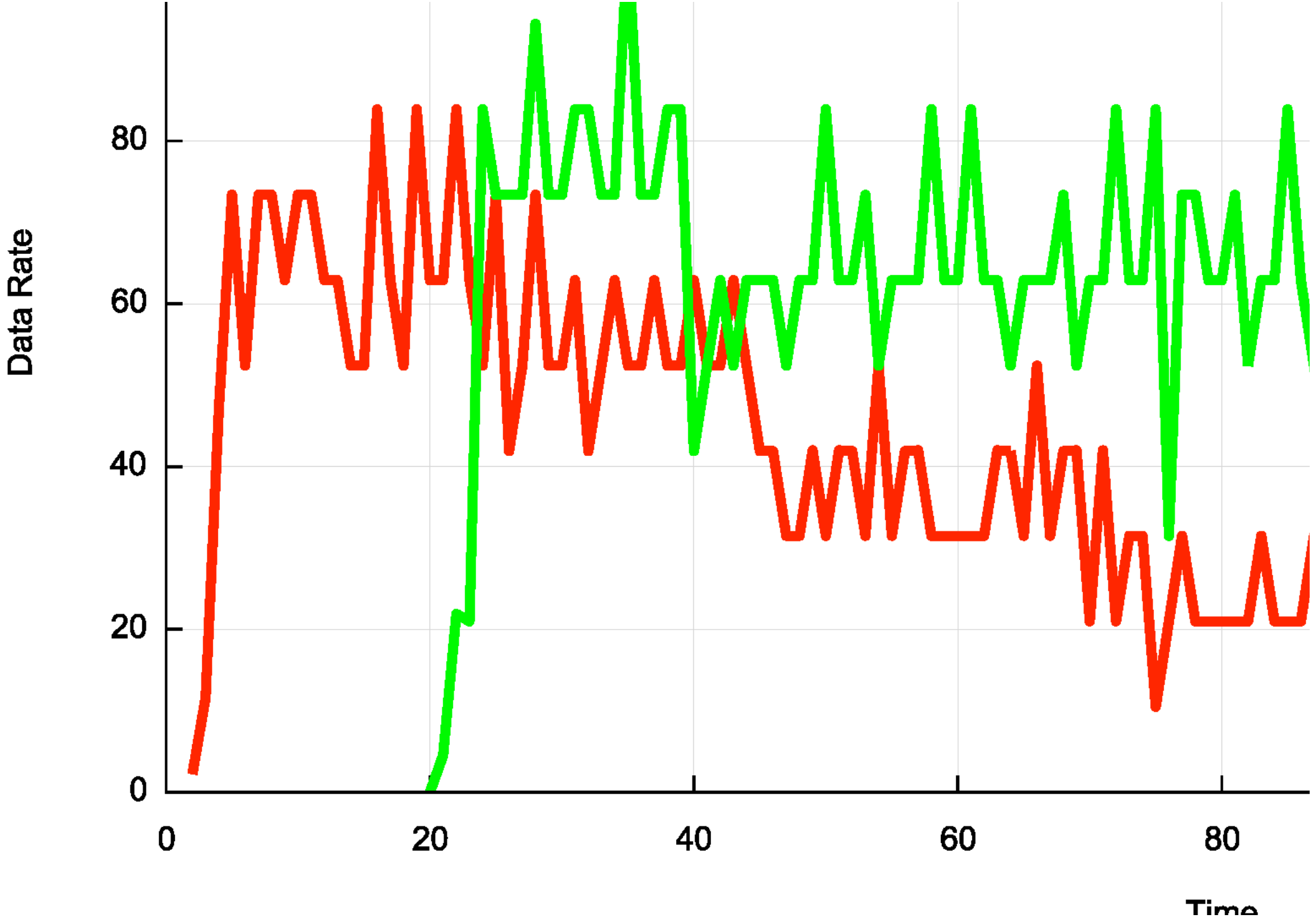

Data Rate
0
20
40
60
80
0
20
40
60
80
Time

Figure C: BBR vs CUBIC fairness (both on TCP) (source)

Another option is of course that a (small) implementation error causes your CCA to perform suboptimally, slowing down your own traffic. Seeing as all these things have to be re-implemented from scratch, I guarantee these kinds of bugs will pop up. Since congestion control can be very tricky to debug, it might be a while before you notice. For example, when working on their original QUIC implementation, Google uncovered an old TCP CUBIC bug and saw major improvements for both TCP and QUIC after fixing it.

Counterarguments

- Networks have mitigations and rate limiting in place to prevent this kind of abuse
- Congestion control manipulation has been possible in TCP since the start and seems not to occur to a problematic degree in practice
- There is no evidence of large players (e.g., Youtube, Netflix) employing this type of strategy to make sure their traffic gets priority
- Really dude, this again? What do you think browsers were doing when they started opening 6 TCP connections per domain?

6. Too soon and too late

QUIC has been around for quite a long time: starting as a Google experiment in 2012 (gQUIC), it was passed on to the IETF for standardization (iQUIC) in 2015 after a decent live deployment at scale, proving its potential. However, even after 6 years of design and implementation, QUIC is far from (completely) ready. The IETF deadline for v1 had already been extended to November 2018 and has now been moved again to July 2019. While most large features have been locked down, even now changes are being made that lead to relatively major implementation iterations. There are over 15 independent implementations, but only a handful implement all advanced features at the transport layer. Even fewer (two at the moment) implement a working HTTP/3 mapping. Since there are major low-level differences between gQUIC and iQUIC, it is as of yet unclear if results from the former will hold true in the latter. This means the theoretical design is maybe almost finished, but implementations remain relatively unproven (though Facebook claims to already be testing QUIC+HTTP/3 for some internal traffic). There is also not a single (tested) browser-based implementation yet, though Apple, Microsoft, Google and Mozilla are working on IETF QUIC implementations and we ourselves have started a POC based on Chromium.

Too (much too) soon

This is problematic because the interest in QUIC is rising, especially after the much talked-about name-change from HTTP-over-QUIC to HTTP/3. People will want to try it out as soon as possible, potentially using buggy and incomplete implementations, in turn leading to sub-par performance, incomplete security and unexpected outages. People will in turn want to debug these issues, and find that there are barely any advanced tools or frameworks that can help with that. Most existing tools are tuned for TCP or don't even look at the transport layer and QUIC's layer-spanning nature will make debugging cross-layer (e.g., combining 0-RTT with H3 server push) and complex (e.g., multipath, forward error correction, new congestion control) issues difficult . This is in my opinion an extensive issue; so extensive that I've written a full paper on it, which you can read here. In it, I advocate for a common logging format for QUIC which allows creating a set of reusable debugging and visualization tools, see Figure **D**.

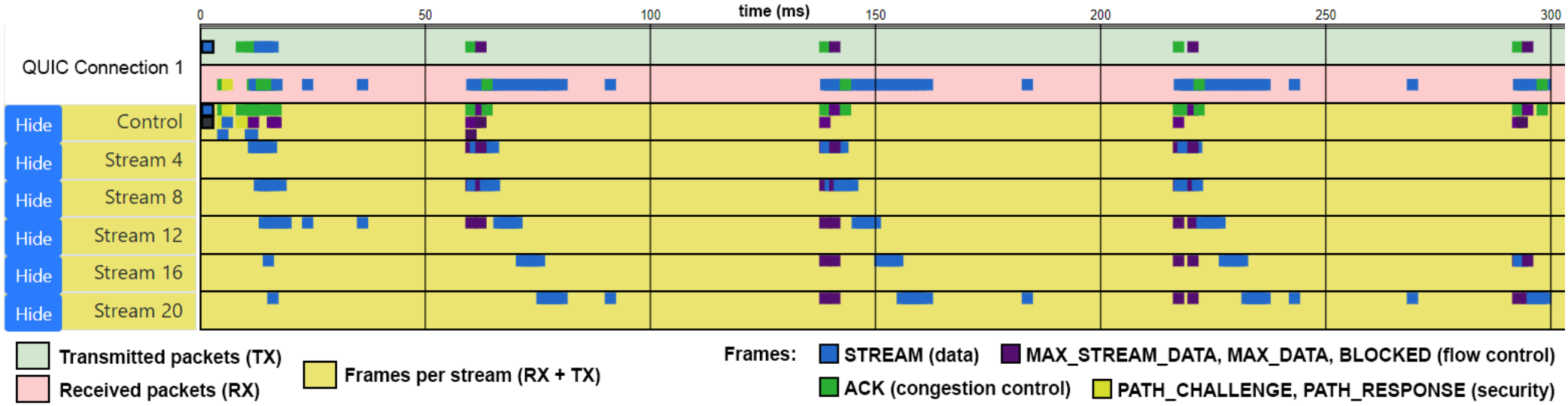


Figure D: A per-stream visualization of a QUIC connection helps see bandwidth distribution and flow control across resources (source)

As such, there is a risk that QUIC and its implementations will not be ready (enough) by the time the people want to start using it, meaning the "Trough of Disillusionment" may come too early and broad deployment will be delayed years. In my opinion, this can also be seen in how CDNs are tackling QUIC: Akamai, for example, decided not to wait for iQUIC and instead has been testing and deploying gQUIC for a while. LiteSpeed burns at both ends of the candle, supporting both gQUIC and pioneering iQUIC. On the other hand though, Fastly and Cloudflare are betting everything on just iQUIC. Make of it what you will.

Too (little too) late

While QUIC v1 might be too early, v2 might come too late. Various advanced features (some of which were in gQUIC), such as forward error correction, multipath and (partial) unreliability are intentionally kept out of v1 to lower the overall complexity. Similarly, major updates to HTTP/3, such as to how cookies work, are left out. In my opinion, H3 is a very demure mapping of HTTP/2 on top of QUIC, with only minor changes. While there are good reasons for this, it means many opportunities for which we might want to use QUIC have to be postponed even longer.

The concept of separating QUIC and HTTP/3 is so that QUIC can be a general-purpose transport protocol, able to carry other application layer data. However, I always struggle to come up with concrete examples for this… WebRTC is often mentioned, and there was a concrete DNS-over-QUIC proposal, but are there any other projects ongoing? I wonder if there would be more happening in this space if some of the advanced features would be in v1. The fact that the DNS proposal was postponed to v2 surely seems to indicate so.

I think it will be difficult to sell QUIC to laymen without these types of new features. 0-RTT sounds nice, but is possibly not hugely impactful, and could be done over TCP. Less Head-of-Line blocking is only good if you have a lot of packet loss. Added security and privacy sounds nice to users, but has little added value besides their main principle. Google touts 3-8% faster searches: is that enough to justify the extra server and setup costs? Does QUIC v1 pack enough of a punch?

Counterarguments

- Browsers will only support QUIC when it's stable (enough) and users probably won't notice most bugs
- Debugging QUIC will be done by professionals who can get by with their own tools and logging formats
- HTTP/2 has had some pretty big issues and bugs (which no-one even seemed to notice for a long time) and yet has found a decent uptake
- Even if QUIC doesn't get a huge uptake in the first two years, it's still worth it. We're in this for the next 30 years.
- QUIC v2 will come soon enough, there are already working groups and proposals looking at unreliability, multipath, WebRTC etc.
- The QUIC working group was never intended to bring major changes to HTTP and that work will continue in the HTTP working group
- QUIC's flexibility ensures that we can now iterate faster on newer features, both on the application and transport layer
- Laymen will follow whatever the big players do (this is how we got into this JavaScript framework mess, remember?)
- A wizard is never late

Conclusion

If you've made it through all that: welcome to the end! Sit, have a drink!

I imagine there will be plenty of different feelings across readers at this point (besides exhaustion and dehydration) and that some QUIC collaborators might be fuming. However, keep in mind what I stated in the beginning: this is me trying to take a "Devil's Advocate" viewpoint, trying to weed out logical errors in arguments pro and con QUIC. Most (all?) of these issues are known to the people who are standardizing QUIC and all their decisions are made after (very) exhaustive discussion and argumentation. I probably even have some errors and false information in my text somewhere, as I'm not an expert on all subtopics (if so, please let me know!). That is exactly why the working groups are built up out of a selection of people from different backgrounds and companies: to try and take as many aspects into consideration as possible. Trade-offs are made, but always for good reasons.

That being said, I still think QUIC might fail. I don't think the chance is high, but it exists. Conversely, I also don't think there is a big chance it will succeed from the start and immediately gain a huge piece of the pie with a broader audience outside of the bigger companies. I think the chance is much higher that it fails to find a large uptake at the start, and that it instead has to gain a broad deployment share more slowly, over a few years. I think this will be slower than what we've seen with HTTP/2, but (hopefully) faster than IPv6.

I personally still believe strongly in QUIC (I should, I'm betting my PhD on it…). It's the first major proposed change on the transport layer that might actually work in practice (the arguments in this post are several times worse and more extensive for many previous options). I feel very grateful to have the chance to witness QUIC's standardization and deployment up close. As it is made to evolve, I think it has all the potential to survive a slower uptake, and remain relevant for decades. The bigger companies will deploy it, debug it, improve it, open source it, and in 5 years time more stuff will be running on QUIC than on TCP.

Thanks to:

- Barry Pollard, writer of the excellent HTTP/2 in action.
- Daniel Stenberg, Daan De Meyer, Loganaden Velvindron, Dmitri Tikhonov, Subodh Iyengar, Mariano Di Martino and other commenters on the original draft of this text.

Custom figures were made with draw.io
Would you like to know more?

Search

Search: Search

Planet Performance

- A project by Stoyan Stefanov
- Powered by WordPress
- Grab the RSS feed
- Part of Planet Performance
- Check out the Podcast
- ... and the Feed
- Tweets @perfplanet
- ... and @stoyanstefanov
- Perfplanet on Facebook

Archives

- 2019
- 2018
- 2017
- 2016
- 2015
- 2014
- 2013
- 2012
- 2011
- 2010
- 2009