Sidecar: In-Network Performance Enhancements in the Age of Paranoid Transport Protocols

Gina Yuan, David K. Zhang, Matthew Sotoudeh, Michael Welzl[†], Keith Winstein
Stanford University and University of Oslo[†]

ABSTRACT

In response to ossification and privacy concerns, post-TCP transport protocols such as QUIC are designed to be "paranoid"—opaque to meddling middleboxes by encrypting and authenticating the header and payload—making it impossible for Performance-Enhancing Proxies (PEPs) to provide the same assistance as before. We propose a research agenda towards an alternate approach to PEPs, creating a *sidecar protocol* that is loosely-coupled to the unchanged and opaque, underlying transport protocol. The key technical challenge to sidecar protocols is how to usefully refer to the packets of the underlying connection without ossification. We have made progress on this problem by creating a tool we call a *quACK (quick ACK)*, a concise representation of a multiset of numbers that can be used to efficiently decode the randomly-encrypted packet contents a sidecar has received. We implement the quACK and discuss how to achieve several applications with this approach: alternate congestion control, ACK reduction, and PEP-to-PEP retransmission across a lossy subpath.

CCS CONCEPTS

• **Networks** → **Network protocols**; *Network protocol design*; *Transport protocols*; **Middle boxes / network appliances**;

KEYWORDS

networks, network protocols, transport protocols, middleboxes, performance-enhancing proxy (PEP), QUIC

ACM Reference Format:
Gina Yuan, David K. Zhang, Matthew Sotoudeh, Michael Welzl[†], Keith Winstein. 2022. Sidecar: In-Network Performance Enhancements in the Age of Paranoid Transport Protocols. In *The 21st ACM Workshop on Hot Topics in Networks (HotNets '22), November 14–15, 2022, Austin, TX, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3563766.3564113

ACM ISBN 978-1-4503-9899-2/22/11...$15.00
https://doi.org/10.1145/3563766.3564113

1 INTRODUCTION

In the 1970s, the Internet's architects created TCP and IP as different beasts. IP is spoken and understood by every host and router. But in the canonical model, TCP is implemented only in hosts [4, 28], while routers and other network components simply exchange IP datagrams on a best-effort basis without regard to their payloads.

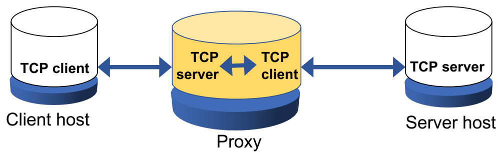
TCP/IP benefited greatly from its end-to-end principles, but in practice, the "right way" to implement TCP can depend on the particulars of the network path—particulars that hosts are typically unaware of. An appropriate retransmission timeout or congestion-control scheme for a heavily multiplexed wired network wouldn't be ideal for paths that include a high-delay satellite link, Wi-Fi with bulk ACKs and frequent reordering, or a cellular WWAN [10, 22]. Moreover, end-to-end retransmissions can be wasteful when a long network path includes a single hop with nontrivial noncongestive loss.

By the 1990s, many networks had broken from the canonical model by deploying in-network TCP accelerators, also known as Performance-Enhancing Proxies [2, 3, 5, 6, 8, 11, 12, 17, 24, 26]. These "PEPs" can insert themselves in the middle of each TCP connection to change the network behavior over a specific subpath (Fig. 1(a)). Because TCP isn't encrypted or authenticated, PEPs achieve this without the cooperation or knowledge of end hosts. A 2011 study estimated that 25% of Internet paths include a TCP PEP [13], and it's likely that many users benefit—especially those on unusual or innovative access networks for which the default congestion-control or retransmission behavior of a faraway server isn't well-tuned.

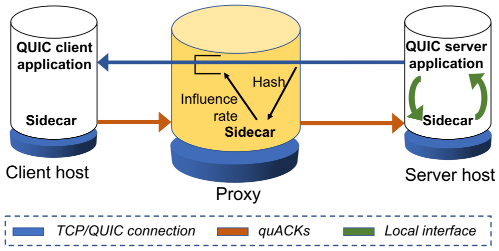
PEPs also carry a big cost: protocol ossification [25]. When a middlebox inserts itself in a connection and enforces its preconceptions about what an IP payload represents, it can thwart the transport protocol's evolution, dropping traffic between hosts that try to speak an upgraded version of the protocol. TCP PEPs have hindered the deployment of new TCP options and behaviors, such as multipath TCP [27].

In response to this ossification, and to an increased emphasis on privacy and security, post-TCP transport protocols are designed to be "paranoid"—opaque and impervious to meddling middleboxes, by encrypting and authenticating the transport header and payload. The most popular of these is QUIC [16], found in billions of deployed Web browsers and millions of webservers [29], as well as encrypted protocols used by applications such as Signal, Zoom, and Mosh.

QUIC's encryption and authentication mean that connections can't be "split" by a middlebox without host cooperation,



(a) Connection splitting with a traditional PEP.



(b) Congestion control division with the sidecar.

Figure 1: Existing PEPs vs. the proposed sidecar. The transport connection is not interrupted in case (b).

nor can middleboxes understand the sequence or acknowledgment numbers in transit. This makes it impossible for a PEP to *transparently* adjust a flow's loss-detection, retransmission, or congestion-control scheme.

What about a "non-transparent" PEP that announces its presence and offers to help? Past work has shown how "paranoid" transport protocols *could* be redesigned to permit PEPs [9, 14]. For a protocol like QUIC, this would probably involve redesigning the system of cryptographic keys and header encryption so that a host could credential a PEP for limited access to some transport headers (e.g., sequence and acknowledgment numbers) without compromising other security properties. This would add considerable complexity and would tightly couple the transport protocol to a PEP's possible needs. We believe this would be a tough row to hoe.

In this paper, we propose a research agenda towards an alternate approach to PEP assistance for next-generation transport protocols. We believe it is possible to leave the transport protocol unchanged on the wire, and instead create a second protocol—a "sidecar" protocol—that is only *loosely coupled* to the underlying transport protocol, and is spoken between a PEP and one or both hosts, or between two PEPs (Fig. 1(b)). PEPs could volunteer their assistance to hosts, and hosts would accept that assistance or not, without credentialing the PEP, without compromising the underlying security and reliability properties of the protocol, and without tightly coupling the sidecar protocol to the underlying host-to-host transport protocol. Several applications could be achieved with this approach, e.g., alternate congestion control, ACK reduction, and PEP-to-PEP retransmission across a lossy subpath.

In our view, the key technical challenge that has to be solved to enable *any* of these sidecar protocols is the following: if sequence numbers are encrypted, and if the PEP has no special credentials to access them, then how can a sidecar protocol usefully *refer to* the packets of the underlying transport connection? More specifically: **how can a PEP efficiently express a "cumulative ACK + selective ACK" over encrypted sequence numbers?**

$$\textbf{Construction: } R \rightarrow \texttt{quACK}$$
$$\textbf{Decoding: } S + \texttt{quACK} \rightarrow S \setminus R$$

Figure 2: QuACK interface, where R and S are the multisets of received and sent elements, respectively.

We have made progress on solving this problem by creating a tool we call a "quACK" (Fig. 2). A quACK is a *concise* representation of a multiset of numbers that correspond to the randomly-encrypted packet headers a sidecar has received. We refer to packets by these random-looking *identifiers*, as opposed to any protocol-level sequence number. Given a quACK and a list of candidate packets a sidecar has sent, the sidecar can *efficiently* determine exactly which subset of those packets have yet to be received.

Designing such a quACK is non-trivial. One strawman solution could, similar to [21, 23], echo the identifier of every received packet to the sender, who calculates a set difference with its sent packets to find the missing packets. This approach uses extraordinary bandwidth. Another strawman returns a hash of a sorted concatenation of all the received packets, and the sender hashes every subset of sent packets of the same size until it finds the correct subset. This approach can easily become computationally infeasible.

In contrast, our algorithm leverages related theoretical work in *straggler identification* [7] to transfer the minimal amount of data for the sender to efficiently identify its subset of missing packets. In particular, for $n = 1000$ sent packets and up to $t = 20$ missing packets, we implement a quACK with the following metrics:

(1) 82 bytes transmitted from the receiver to the sender,
(2) ≈ 100 ns additional processing time per packet,
(3) < 100 us decoding time from quACK and list of candidate packets to the missing packets,
(4) 0.000023% chance that a candidate packet has an indeterminate result.

In the rest of the paper, we describe three sidecar protocols that leverage the quACK to enable performance enhancements for next-generation transport protocols (§ 2). We discuss the quACK primitive and its algorithms in more detail (§ 3). We demonstrate that our implementation of the quACK can meet the practical performance constraints of our sidecar protocols (§ 4). Finally, we discuss a research agenda and conclude (§ 5).

2 SIDECAR PROTOCOLS

To prevent the same ossification that emerged from existing PEPs, *proxies* on a connection's path should act as regular routers for packets between the *end hosts*—they can withhold or delay packets, but they cannot modify the packets or make decisions based on their contents. The *server* end host primarily sends data to the *client* end host.

Name	Proxy Role	Server Role	Client Role
Congestion Control Division	Send and receive quACKs. Determine the downstream sending rate.	Receive quACKs. Determine the congestion window.	Send quACKs.
ACK Reduction	Send quACKs.	Receive quACKs. Move the sending window.	None.
In-Network Retransmission	Send and receive quACKs. Buffer and retransmit packets when necessary. Set the communication frequency based on the loss ratio.	None.	None.

Table 1: Example proposed sidecar protocols.

To enable performance enhancements, we propose that hosts and proxies have separate *sidecars* that communicate with other sidecars via the *sidecar protocol* to influence the E2E-encrypted *base protocol*, without ossification. In particular, proxies should still be regular routers, although end hosts can update the base protocol due to their explicit consent.

Sidecars communicate with each other by sending *quACKs (quick ACKs)* (Fig. 2). They can also configure sidecar protocol parameters with each other such as the communication frequency and properties of the quACK. We describe the quACK primitive and these algorithms in detail in § 3.

A key contribution of the quACK is that it can express the same information as cumulative and selective TCP ACKs *without* any visiblity into the raw, protocol-level sequence numbers. We further show that with just this information, we can provide three different performance enhancements to QUIC as sidecar protocols that do not rely on specific protocol fields (Table 1). These protocols demonstrate that quACKs can enable PEP functionality previously thought to be impossible in the E2E-encrypted setting, opening up a research agenda for in-network acceleration without ossification.

2.1 Congestion Control Division

Servers can sometimes transmit data through a PEP faster than to the client directly, and splitting an end-to-end connection into multiple segments enables the PEP to better adjust its sending rate or implement a different kind of congestion control on each segment entirely. However, PEPs cannot spoof and split connections that use QUIC, and proposed QUIC standards [19, 20] cannot be applied generally to other E2E-encrypted protocols, unlike the sidecar.

QuACKs make it possible to perform PEP-style connection-splitting for congestion control via a sidecar protocol, *even when the base protocol is E2E-encrypted*. In Fig. 1(b), the client sends quACKs to the proxy, which separately sends quACKs to the server. On each segment, a quACK can be sent e.g., at a fixed interval such as once per RTT. The quACK enables the sender on each segment to determine exactly which packets have yet to be received since the last quACK.

The sidecars use information derived from quACKs to influence the sending rate on the downstream segment. For example, the proxy can drain a buffer of unforwarded QUIC packets at a slower rate if it detects a large number of packets have yet to be received. The server end host, which can modify the base protocol, can decrease the congestion window (cwnd). The server no longer needs to rely on end-to-end ACKs to make decisions to increase the cwnd, though these ACKs still govern the retransmission logic.

The only changes that need to be made to the end hosts are installing a library on the client to generate quACKs, and on the server to decode quACKs and adjust the cwnd. No changes need to be made to the QUIC protocol specification.

2.2 ACK Reduction

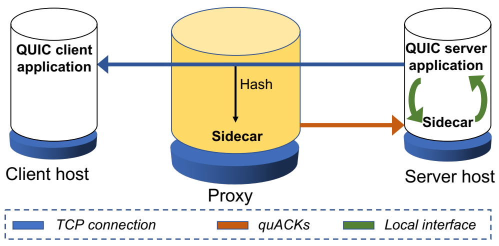


Figure 3: Sidecar ACK reduction.

A TCP PEP proposal for ACK reduction [18] drastically reduces the number of ACKs a client needs to transmit by having the PEP acknowledge packets on behalf of the client. A strawman proxy for E2E-encrypted protocols could reflect every packet it receives from the server, but this is even worse than client ACKs, which can accumulate sequence numbers. Fig. 3 shows how a proxy sidecar could provide the same functionality for E2E-encrypted protocols by using quACKs, which are agnostic of protocol-level sequence numbers but can concisely represent the received packets.

The sidecar protocol effectively treats the quACKs as client ACKs. The proxy can send quACKs, e.g., every other packet such as in TCP, much more frequently than in the protocol for congestion control (§ 2.1). The proxy does not need to read or modify QUIC packet contents, and the client does not need to participate in the sidecar protocol at all. This protocol can enable the server to move its sending window ahead more quickly than if it had to wait for ACKs from the client an additional hop away. The client can also transmit fewer ACKs using the proposed ACK frequency extension in QUIC [15], reducing network congestion.

Though these quACKs generally replace ACKs from the client, end-to-end ACKs have some special roles that the sidecar protocol cannot fulfill. For example, end-to-end ACKs may convey Explicit Congestion Notification (ECN) information. Also, quACKs do not inform about packets that are dropped from the proxy to the client. As a solution, the server can still rely on quACKs in most cases, and use the less frequent end-to-end ACKs when retransmission is necessary.

2.3 In-Network Retransmission

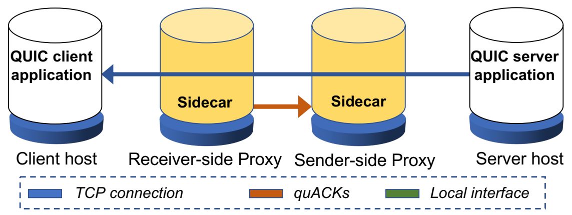


Figure 4: In-network retransmission with the sidecar.

In Fig. 4, sidecar instances on two routers are statically configured to retransmit packets in case packet loss happens on the path segment between them; such a mechanism was recently proposed to the IETF [23]. In-network retransmission can be beneficial when the RTT between the two routers is significantly smaller than the end-to-end RTT [1]. The proposed mechanism uses per-packet hashes to ACK every packet, similar to the strawman proxy, adding unnecessarily large overhead. Different from quACKs, such hashes are not cumulative, and hence lost ACKs are not accounted for.

The receiver-side proxy on the left-hand side of Fig. 4 transmits a quACK to the sender-side proxy on the right-hand side. The sender-side proxy does not need to read or modify packet contents, just hold packets in a buffer in case they need to be retransmitted. The interval at which the receiver-side proxy produces and transmits the quACK is flexible, as it should ideally depend on the loss ratio. The sender-side proxy determines the loss ratio, and can configure the communication frequency accordingly.

3 QUACK

As the previous section has illustrated, the senders and receivers of quACKs in a sidecar protocol generally need to be able to identify packets, *concisely* acknowledge their reception, and *efficiently* decode the acknowledgments—but we were previously vague about what the algorithms looked like.

We now elaborate on these algorithms. In this section we describe a construction of the quACK based on solving a system of power sum polynomials. Our construction is adapted from a solution to the related problem of *straggler identification* [7]. After presenting the algorithm, we discuss how senders and receivers would practically apply it to transmit and decode quACKs in sidecar protocols.

3.1 The QuACK Problem and Power Sums

We describe the quACK problem and a solution based on solving a system of power sum polynomial equations.

Let a sender send a multiset of elements (which are integers) S to a receiver. At any given time the receiver has received a subset $R \subseteq S$ of the sent elements. We would like the receiver to communicate a small amount of information to the sender, who then efficiently decodes the missing elements, i.e., the set difference $S \setminus R$. We call this small amount of information the "quACK", and the problem is: **what is in a quACK and how do we decode it?**

Consider the simplest case, when the receiver is only missing a single element. The receiver can communicate the sum $\sum_{x \in R} x$ of the received elements to the sender. The sender computes the sum $\sum_{x \in S} x$ of the sent elements and subtracts the sum from the receiver, calculating:

$$\sum_{x \in S} x - \sum_{x \in R} x = \sum_{x \in S \setminus R} x,$$

which is the sum of elements in the set difference. In this case, the sum is exactly the value of the missing element.

In fact, we can generalize this scheme to any number of missing elements m. Instead of transmitting only a single sum, the receiver communicates the first m *power sums* to the sender, where the i-th power sum of a multiset R is defined as $\sum_{x \in R} x^i$. The sender then computes the first m power sums of S and calculates the respective differences d_i for $i \in [1, m]$, producing the following system of equations:

$$\left\{ \sum_{x \in S \setminus R} x^i = d_i \mid i \in [1, m] \right\}.$$

Efficiently solving these m power sum polynomial equations in m variables is a well-understood algebra problem [7]. The solutions to these equations are exactly $x \in S \setminus R$.

Since the receiver does not know m ahead of time, both parties can also maintain a count of elements accumulated in the power sums since the beginning of the connection. To avoid two rounds of communication to first determine m and then calculate the m power sums to send, the receiver can instead communicate a quACK that contains t power sums for some threshold $t \geq m$, and the count. The sender can then determine m based on its own count and use the first m power sums. Note that if $t < m$, the decoding process fails.

3.2 Using QuACKs in a Sidecar Protocol

In a sidecar protocol, a sender sends a multiset of packets S and a receiver receives a subset $R \subseteq S$. We think of packets as numbers, e.g., 32 bits from a randomly-encrypted QUIC header, and call these numbers the *identifiers*. The receiver communicates a quACK, which consists of t power sums for some threshold t, and a count of accumulated packets, to the sender. The sender then decodes the packets in $S \setminus R$ using the technique described above.

The receiver may configure several protocol parameters:

(1) a threshold number of missing packets t,
(2) the number of bits b used in the identifier,
(3) the communication frequency of quACKs.

Receivers select t based on the communication frequency, and the estimated bandwidth usage and loss rate on the link.

When the sidecar protocol is initiated, both the sender and receiver initialize t power sums to 0. To reduce decoding time, we amortize power sum calculation: The sender updates the sums before sending each packet, and the receiver updates them when receiving each packet. Both parties also maintain a count, and the sender maintains a log of sent packets. To bound the size of the quACK while preserving a unique solution, all power sum arithmetic is performed modulo the largest prime that can be expressed in b bits.

When the receiver is ready to send a quACK, it sends the $b \cdot t$ bits corresponding to its t power sums, and the count, to the sender. The sender subtracts the received count from its own count to determine the number of missing packets m. Note that the number of bits used to represent the count only needs to be big enough to represent this difference, and the count itself can wraparound. If the difference also wraps around, then the polynomial equations either cannot be solved or the solutions do not correspond to packets in S.

The sender decodes the missing packets in its log by solving the first m polynomial equations derived from the quACK, mapping the identifiers to their original packets. If b is too small, a decoded identifier may correspond to multiple candidate missing packets. The sender considers the fate of these packets indeterminate, and interprets the results based on the specific sidecar protocol. If $t < m$, decoding fails because there are not enough equations to solve.

3.3 Practical Considerations

There are some practical considerations when using a quACK and setting parameters. We discuss how small modifications to the quACK can address these considerations.

Resetting the threshold. The threshold parameter should only apply to the number of missing packets since the last received quACK, rather than over the entire connection. When decoding missing packets, the sender assumes they will never be received and removes them from its log and power sums. Thus these packets will not be counted in the threshold of the next received quACK.

Re-ordered packets. Packets may also be re-ordered, causing missing packets to later be received. Thus discarding missing packets can be problematic. The sender can buffer missing packets for a period of time before actually deleting them from the log to allow the missing packet to be received.

In-flight packets. The sender may have logged many more packets since when the receiver initially transmitted the quACK. Say the sender has logged n' packets and the quACK includes n packets where $n' - n > t$. Rather than increasing t, the sender can temporarily truncate the log suffix such that the log has length $n + t$, considering the truncated packets to be in transit. When the sender decodes the remaining log, it considers any continuous suffix of missing packets to also be in transit, instead of actually missing.

Exceeding the threshold. If the number of missing packets exceeds the threshold, the sender and receiver must reset the connection if they wish to use the quACK.

Dropped quACKs. The implementation is resilient to quACKs that are dropped in transmission, since the power sums in both the sender and receiver are cumulative.

4 EVALUATION

We demonstrate that our implementation of the quACK based on power sums can both *concisely* represent and *efficiently* decode the set of received packets from a list of sent packets. Our code is shared at https://github.com/ygina/quack.

We evaluate our implementation on a 2019 MacBook Pro running macOS Monterey v12.4 with a 2.4 GHz 8-Core Intel i9 processor and 32 GB memory, representative of a client end host. Our code, including all benchmarking code and the two strawman solutions, is written in 1408 lines of C++.

A quACK that represents $n = 1000$ sent packets and up to $t = 20$ missing packets with $b = 32$-bit identifiers takes 106 us to construct and 61 us to decode, and requires 82 bytes to be transferred from the receiver to the sender. Using 32-bit identifiers, there is a 0.000023% chance that a candidate packet has an indeterminate result. In comparison, our two strawmans use extraordinary bandwidth or computation.

We additionally show how modifying 1) the threshold number of missing packets t and 2) the number of identifier bits b affects these metrics, and discuss how an end host could select these parameters and 3) the communication frequency that affects n based on the specific sidecar protocol.

4.1 Comparison to Strawman QuACKs

	Construction Time	Decoding Time	QuACK Size (bits)
Strawman 1	222 us	126 us	$b \cdot n = 32000$
Strawman 2	387 ns	$\approx$7e+06 days	$256 + c = 272$
Power Sums	106 us	61 us	$t \cdot b + c = 656$

Table 2: Strawmans compared to the power sum QuACK, using $n = 1000$, $t = 20$, and $c = 16$ bits to store the count. All use $b = 32$-bit identifiers, which results in a 0.000023% chance that a candidate packet has an indeterminate result. Average of 100 trials with warmup.

The strawman quACKs described in § 1 are unrealistic, using either extraordinary space or computation (Table 2). In comparison, the size of our power sum quACK is 82 bytes (4000 bytes in Strawman 1), and is proportional only to the threshold. The decoding time of the power sum quACK is 61 us ($\approx$7e+06 days in Strawman 2).

In the following sections, we discuss how modifying the quACK parameters t, n, and b affects these metrics, and argue how these results put the quACK within the latency and bandwidth constraints of our three sidecar protocols.

4.2 Configuring QuACK Parameters

Construction Time The construction time is how long it takes to construct a quACK (with threshold t) from a list of n packets with b-bit identifiers. Typically, the construction

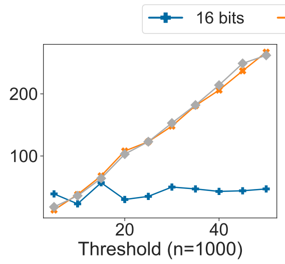


Figure 5: Construction (us).

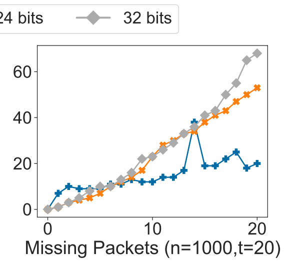


Figure 6: Decoding (us).

time is directly proportional to t, as it uses one modular multiplication and addition per end host and for each power sum determined by t (Fig. 5). The value of b determines which hardware instructions and, in the 16-bit case, pre-computation optimizations the arithmetic can use. Our implementation amortizes construction time (≈ 100 ns per packet) by updating the power sums as each packet is sent and received.

Decoding Time The decoding time is how long it takes to decode m missing packets from a list of n packets with b-bit identifiers, given a quACK. It is directly proportional to m, which is at most t (Fig. 6). Decoding involves calculating the coefficients of a polynomial with degree m, and solving the polynomial. For a small n, such as here, it is more efficient to plug in all candidate roots than to solve the roots directly. Again, b determines the hardware and pre-computation optimizations. We expect stable links to mostly not be missing packets, which takes virtually no time to decode.

QuACK Size The quACK size is the number of bits transmitted from the receiver to the sender. Our implementation uses $b \cdot t + c = 656$ bits, where c is the number of bits used to hold the number of missing packets. Note c should be at least $\log_2 t$, or a little larger to avoid having to detect wraparound.

Collision Probability The collision probability is the probability that a randomly-chosen b-bit identifier in a list of n packets maps to more than one packet in that list. Thus if packets with that identifier have both been received and dropped, the fate of those packets is considered indeterminate. Note that collisions are known to the sender beforehand. If we assume that identifiers are randomly-distributed, which is the case in randomly-encrypted QUIC packet headers, this probability is equal to $1-(1-1/2^b)^{n-1}$. The more bits we use, the more likely we are able to disambiguate actual missing packets from their candidates. When $n = 1000$, using 32 bits results in an almost negligible chance of collision while using 16 bits results in a 1.5% chance (Table 3).

Identifier Bits	8	16	24	32
Collision Prob.	0.98	0.015	6.0e-05	2.3e-07

Table 3: Collision probabilities for $n = 1000$.

4.3 Selecting the Communication Frequency

We discuss how a sender might select a communication frequency for each of our three proposed sidecar protocols.

In congestion control division, which unlike TCP does not ACK for reliability, we quACK only once per RTT. Assuming a 60ms RTT on a 200 Mbps link and a maximum handled 2% loss rate, at 1500 bytes/packet (a typical MTU), this is ≈ 1000 sent packets with 20 missing packets per RTT, the parameters in §4.1. The added latency of the quACK is the amortized construction time, or ≈ 100 ns per packet.

In ACK reduction, the receiver could quACK e.g., every $n = 32$ packets, similar to TCP which ACKs every other packet. However, the quACK works for encrypted protocols. To reduce the quACK size, we can omit c, which is always n. Setting $t < n$ uses less bandwidth compared to Strawman 1.

The quACK frequency for in-network retransmission should change dynamically based on the loss ratio. The sender who configures this frequency could target a constant $t = 20$ missing packets per quACK. If the link is relatively stable, the sender-side proxy could decrease the frequency through the sidecar protocol, which the receiver-side proxy applies by the subsequent RTT. Only n changes per quACK, and for large n, we can use the decoding algorithm that depends only on t.

5 CONCLUSION

Sidecar protocols provide PEP-like functionality to "paranoid" transport protocols, without ossification. We describe a primitive called a *quACK (quick ACK)* that allows a middlebox to concisely and efficiently acknowledge an arbitrary subset of E2E-encrypted packets, enabling sidecar protocols.

For sidecar protocols to become a reality, several questions must still be answered: How do we handle adversarial proxies? How does an end host discover participating proxies, and how would a proxy interact with multipath transport protocols? How do we transition in-network accelerators from the old world of TCP to a new world dominated by "paranoid" and opaque transport protocols? How do we build client libraries that speak the sidecar protocol for a variety of end hosts including mobile phones, browsers, and IoT devices?

There are also questions that explore the newly-proposed functional separation: What other sidecar protocols could a quACK enable, and what similar protocol-agnostic digests could we design to provide even more enhancements? How do we further optimize the algorithm and implementation of the quACK towards nearly-zero overhead quACKing? How will transport protocols continue to evolve with increasing privacy and ossification concerns, and how should the role of in-network accelerators change in response? Answering such questions will be essential in designing next-generation transport protocols and middlebox functionality.

ACKNOWLEDGMENTS

We thank Mary Wootters for valuable discussions on error correcting codes. This work was supported in part by NSF grants 2045714, 2039070, 2028733, 1931750, 1918056, 1763256, and DGE-1656518, DARPA contract HR001120C0107, a Stanford Graduate Fellowship, a Sloan Research Fellowship, and by Google, Huawei, VMware, Dropbox, Amazon, and Meta Platforms.

REFERENCES

[1] Runa Barik, Michael Welzl, Peyman Teymoori, Safiqul Islam, and Stein Gjessing. 2020. Performance Evaluation of In-network Packet Retransmissions using Markov Chains. In *2020 International Conference on Computing, Networking and Communications (ICNC)*. 10–16. https://doi.org/10.1109/ICNC47757.2020.9049757

[2] J. Border, M. Kojo, J. Griner, G. Montenegro, and Z. Shelby. 2001. Performance Enhancing Proxies Intended to Mitigate Link-Related Degradations. RFC 3135. (June 2001). https://doi.org/10.17487/RFC3135

[3] C. Caini, R. Firrincieli, and D. Lacamera. 2006. PEPsal: a Performance Enhancing Proxy designed for TCP satellite connections. In *2006 IEEE 63rd Vehicular Technology Conference*, Vol. 6. https://doi.org/10.1109/VETECS.2006.1683339

[4] D. Clark. 1988. The Design Philosophy of the DARPA Internet Protocols. In *Symposium Proceedings on Communications Architectures and Protocols (SIGCOMM '88)*. Association for Computing Machinery, New York, NY, USA, 106–114. https://doi.org/10.1145/52324.52336

[5] Bryce Cronkite-Ratcliff, Aran Bergman, Shay Vargaftik, Madhusudhan Ravi, Nick McKeown, Ittai Abraham, and Isaac Keslassy. 2016. Virtualized Congestion Control. In *SIGCOMM*. ACM, New York, NY, USA, 14.

[6] Paul Davern, Noor Nashid, Cormac J. Sreenan, and Ahmed H. Zahran. 2011. HTTPEP: a HTTP Performance Enhancing Proxy for Satellite Systems. *Int. J. Next Gener. Comput.* 2, 3 (2011).

[7] David Eppstein and Michael T. Goodrich. 2011. Straggler Identification in Round-Trip Data Streams via Newton's Identities and Invertible Bloom Filters. *IEEE Trans. Knowl. Data Eng.* 23, 2 (2011), 297–306. https://doi.org/10.1109/TKDE.2010.132

[8] Viktor Farkas, Balázs Héder, and Szabolcs Nováczki. 2012. A Split Connection TCP Proxy in LTE Networks. In *18th European Conference on Information and Communications Technologies (EUNICE)*, Róbert Szabó and Attila Vidács (Eds.), Vol. LNCS-7479. Springer, Budapest, Hungary. https://doi.org/10.1007/978-3-642-32808-4_24

[9] Bryan Ford and Janardhan R. Iyengar. 2008. Breaking Up the Transport Logjam. In *7th ACM Workshop on Hot Topics in Networks - HotNets-VII, Calgary, Alberta, Canada, October 6-7, 2008*, Carey L. Williamson, David G. Andersen, and Steve D. Gribble (Eds.). ACM SIGCOMM, 85–90. http://conferences.sigcomm.org/hotnets/2008/papers/15.pdf

[10] Prateesh Goyal, Mohammad Alizadeh, and Hari Balakrishnan. 2017. Rethinking Congestion Control for Cellular Networks. In *Proceedings of the 16th ACM Workshop on Hot Topics in Networks (HotNets-XVI)*. Association for Computing Machinery, New York, NY, USA, 29–35. https://doi.org/10.1145/3152434.3152437

[11] D. A. Hayes, D. Ros, and Ö. Alay. 2019. On the importance of TCP splitting proxies for future 5G mmWave communications. In *IEEE LCN*.

[12] Keqiang He et al. 2016. AC/DC TCP: Virtual Congestion Control Enforcement for Datacenter Networks. In *SIGCOMM*. ACM, New York, USA, 14.

[13] Michio Honda, Yoshifumi Nishida, Costin Raiciu, Adam Greenhalgh, Mark Handley, and Hideyuki Tokuda. 2011. Is It Still Possible to Extend TCP?. In *Proceedings of the 2011 ACM SIGCOMM Conference on Internet Measurement Conference (IMC '11)*. Association for Computing Machinery, New York, NY, USA, 181–194. https://doi.org/10.1145/2068816.2068834

[14] Janardhan Iyengar and Bryan Ford. 2009. Flow Splitting with Fate Sharing in a Next Generation Transport Services Architecture. (2009). https://doi.org/10.48550/ARXIV.0912.0921

[15] Jana Iyengar and Ian Swett. 2021. *QUIC Acknowledgement Frequency*. Internet-Draft draft-ietf-quic-ack-frequency-01. Internet Engineering Task Force. https://datatracker.ietf.org/doc/html/draft-ietf-quic-ack-frequency-01 Work in Progress.

[16] Jana Iyengar and Martin Thomson. 2021. QUIC: A UDP-Based Multiplexed and Secure Transport. RFC 9000. (May 2021). https://doi.org/10.17487/RFC9000

[17] Aman Kapoor, Aaron Falk, Theodore Faber, and Yuri Pryadkin. 2005. Achieving faster access to satellite link bandwidth. In *Proceedings IEEE 24th Annual Joint Conference of the IEEE CS and ComSoc.*, Vol. 4. IEEE.

[18] Dzmitry Kliazovich, Simone Redana, and Fabrizio Granelli. 2012. Cross-Layer Error Recovery in Wireless Access Networks: The ARQ Proxy Approach. *Int. J. Commun. Syst.* 25, 4 (apr 2012), 461–477. https://doi.org/10.1002/dac.1271

[19] Mike Kosek, Tanya Shreedhar, and Vaibhav Bajpai. 2021. Beyond QUIC v1: A First Look at Recent Transport Layer IETF Standardization Efforts. *IEEE Communications Magazine* 59, 4 (2021), 24–29. https://doi.org/10.1109/MCOM.001.2000877

[20] Zsolt Krämer, Mirja Kühlewind, Marcus Ihlar, and Attila Mihály. 2021. Cooperative Performance Enhancement Using QUIC Tunneling in 5G Cellular Networks. In *Proceedings of the Applied Networking Research Workshop (ANRW '21)*. Association for Computing Machinery, New York, NY, USA, 49–51. https://doi.org/10.1145/3472305.3472320

[21] Zsolt Krämer, Sándor Molnár, Marcus Pieskä, and Attila Mihály. 2020. A Lightweight Performance Enhancing Proxy for Evolved Protocols and Networks. In *2020 IEEE 25th International Workshop on Computer Aided Modeling and Design of Communication Links and Networks (CAMAD)*. 1–6. https://doi.org/10.1109/CAMAD50429.2020.9209304

[22] Nicolas Kuhn, François Michel, Ludovic Thomas, Emmanuel Dubois, Emmanuel Lochin, Francklin Simo, and David Pradas. 2021. QUIC: Opportunities and threats in SATCOM. *International Journal of Satellite Communications and Networking* (2021). https://doi.org/10.1002/sat.1432 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/sat.1432

[23] Yizhou Li, Xingwang Zhou, Mohamed Boucadair, Jianglong Wang, and Fengwei Qin. 2020. *LOOPS (Localized Optimizations on Path Segments) Problem Statement and Opportunities for Network-Assisted Performance Enhancement*. Internet-Draft draft-li-tsvwg-loops-problem-opportunities-06. Internet Engineering Task Force. https://datatracker.ietf.org/doc/html/draft-li-tsvwg-loops-problem-opportunities-06 Work in Progress.

[24] A. Mihály et al. 2017. Supporting multi-domain congestion control by a lightweight PEP. In *2017 IINTEC*.

[25] G. Papastergiou et al. 2017. De-Ossifying the Internet Transport Layer: A Survey and Future Perspectives. *IEEE Communications Surveys Tutorials* (2017).

[26] Michele Polese, Marco Mezzavilla, Menglei Zhang, Jing Zhu, Sundeep Rangan, Shivendra Panwar, and Michele Zorzi. 2017. milliProxy: A TCP proxy architecture for 5G mmWave cellular systems. In *2017 51st Asilomar Conference on Signals, Systems, and Computers*. 951–957. https://doi.org/10.1109/ACSSC.2017.8335489

[27] Costin Raiciu, Christoph Paasch, Sebastien Barre, Alan Ford, Michio Honda, Fabien Duchene, Olivier Bonaventure, and Mark Handley. 2012. How Hard Can It Be? Designing and Implementing a Deployable Multipath TCP. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI'12)*. USENIX Association, USA, 29.

[28] J. H. Saltzer, D. P. Reed, and D. D. Clark. 1984. End-to-End Arguments in System Design. *ACM Trans. Comput. Syst.* 2, 4 (nov 1984), 277–288. https://doi.org/10.1145/357401.357402

[29] Johannes Zirngibl, Philippe Buschmann, Patrick Sattler, Benedikt Jaeger, Juliane Aulbach, and Georg Carle. 2021. It's over 9000: Analyzing Early QUIC Deployments with the Standardization on the Horizon. In *Proceedings of the 21st ACM Internet Measurement Conference*. Association for Computing Machinery, New York, NY, USA, 261–275. https://doi.org/10.1145/3487552.3487826