Sidekick: In-Network Assistance for Secure End-to-End Transport Protocols

Gina Yuan
Stanford University

Matthew Sotoudeh
Stanford University

David K. Zhang
Stanford University

Michael Welzl
University of Oslo

David Mazières
Stanford University

Keith Winstein
Stanford University

Abstract

In response to concerns about protocol ossification and privacy, post-TCP transport protocols such as QUIC and WebRTC include end-to-end encryption and authentication at the transport layer. This makes their packets opaque to middleboxes, freeing the transport protocol to evolve but preventing some in-network innovations and performance improvements. This paper describes *sidekick protocols*: an approach to in-network assistance for opaque transport protocols where in-network intermediaries help endpoints by sending information adjacent to the underlying connection, which remains opaque and unmodified on the wire.

A key technical challenge is how the sidekick connection can efficiently refer to ranges of packets of the underlying connection without the ability to observe cleartext sequence numbers. We present a mathematical tool called a *quACK* that concisely represents a selective acknowledgment of opaque packets, without access to cleartext sequence numbers.

In real-world and emulation-based evaluations, the sidekick improved performance in several scenarios: early retransmission over lossy Wi-Fi paths, proxy acknowledgments to save energy, and a path-aware congestion-control mechanism we call *PACUBIC* that emulates a "split" connection.

1 Introduction

In the Internet's canonical model, transport is end-to-end and implemented only in hosts. Traditionally, routers and other network components forwarded IP datagrams without regard to their payloads or flow membership [12, 58]; only hosts thought about connections, reliable delivery, or flow-by-flow congestion control.

In practice, however, the best behavior for a transport protocol depends on the particulars of the network path. An appropriate retransmission 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 [25, 42].

By the 1990s, many networks had broken from the canonical model by deploying in-network TCP accelerators, also known as "performance-enhancing proxies" (PEPs) [26]. TCP PEPs can split an end-to-end connection into multiple concatenated connections [10, 17, 23, 28, 34], buffer and retransmit packets over a lossy link [2, 55], virtualize congestion control [14, 29, 49], resegment the byte stream, and enable forward error correction, explicit congestion notification, or other segment-specific enhancements. Because TCP isn't encrypted or authenticated, PEPs can achieve this transparently, without the knowledge or cooperation of end hosts. Roughly 20–40% of Internet paths cross at least one TCP PEP [21, 30].

While many flows benefit from PEPs, their use carries a cost: protocol ossification [21, 53]. When a middlebox inserts itself in a connection and enforces its preconceptions about the transport protocol, it can thwart the protocol's evolution, dropping traffic that uses an upgraded version or new options. TCP PEPs have hindered or complicated the deployment of many TCP improvements, such as ECN++, tcpcrypt, TCP extended options, and multipath TCP [30, 46, 56].

In response to this ossification, and to an increased emphasis on privacy and security, post-TCP transport protocols have been designed to be impervious to meddling middleboxes, by encrypting and authenticating the transport header. We call these newer transport protocols "opaque." The most prevalent is QUIC [32], found in billions of installed Web browsers and millions of servers [68]; other opaque transport protocols are used in WebRTC/SRTP [54], Zoom [69], BitTorrent [4], and Mosh/SSP [63].

This opacity means that middleboxes can't interpose themselves on a connection or understand the sequence numbers of packets in transit. This prevents PEPs from providing assistance, reducing—in some situations—the performance of opaque transport protocols [6, 7, 38, 42, 47]. It's possible to co-design protocols and PEPs to preserve security and privacy while permitting assistance from credentialed middleboxes [19, 24, 33, 59], but challenging to do so without tightly coupling these components, risking ossification and fragility.

In this paper, we propose a method for in-network assistance of opaque transport protocols that tries to resolve this tension. Our approach leaves the transport protocol unchanged on the wire: a secure end-to-end connection between hosts, opaque to middleboxes and free to evolve. No PEPs are credentialed to decrypt the transport protocol's headers.

Instead, we propose a second protocol to be spoken on an adjacent connection between an end host and a PEP. We call this the **sidekick protocol**, and its contents are *about* the packets of the underlying, or "base," connection. Sidekick PEPs assist end hosts by reporting what they've observed about the

packets of the opaque base connection, without coupling their assistance to the details of the base protocol. End hosts use this information to influence decisions about how and when to send or resend packets on the base connection, approximating some of the performance benefits of traditional PEPs. A similar functional separation was first proposed by [67], but this paper presents the first concrete realization of the idea and its nuanced interactions with real transport protocols.

One key technical challenge with this approach is how the sidekick can efficiently refer to ranges of packets in an opaque base connection. These packets appear random to the middlebox, and referring to a range of, e.g., 100 opaque packets in the presence of loss and reordering is not as simple as saying "up to 100" when there are cleartext sequence numbers. In Section 3, we present and evaluate a mathematical tool called a **quACK** that concisely represents a selective acknowledgment of opaque, randomly identified packets. The quACK is based on the insight that we can model the problem as a system of power sum polynomial equations if there is a practical bound on the maximum number of "holes" among the packets being ACKed. We created an optimized implementation [65], building on related theoretical work [22, 35, 50].

A second challenge is how the end host should use information from a sidekick connection to obtain a performance benefit for its base connection. Since the performance benefit comes from changing behavior at the end host rather than the middlebox, transport protocols need to incorporate this information into their existing algorithms for, e.g., loss detection and retransmission, which have gotten increasingly complex over time. To explore this, we designed a sidekick protocol we call Robin, and implemented it in three scenarios:

- A low-latency audio stream over an Internet path that includes a Wi-Fi path segment (low latency with loss), followed by a WAN path segment (higher latency with low loss). Can the sidekick PEP reduce the de-jitter buffer delay by triggering earlier retransmissions on loss?
- An upload over the same path. Can an opaque transport protocol like QUIC, aided by a sidekick PEP at the point between these two path segments, match the throughput of TCP over a connection-splitting PEP?
- A battery-powered receiver, downloading data from the Internet over Wi-Fi. If the Wi-Fi access point sends sidekick quACKs on behalf of the receiver, can it reduce the number of times the receiver's radio needs to wake up to send an end-to-end ACK?

A third technical challenge is how knowledge about *where* loss occurs along a path should influence a congestion-control scheme. The challenge in any such scheme is how to maximize the congestion window while sharing the network fairly with competing flows. We present a path-aware modification to the CUBIC congestion-control algorithm [27], which we call **PACUBIC**, that approximates the congestion-control behavior of a PEP-assisted split TCP CUBIC connection while making its decisions entirely on the host.

Summary of results. Concretely realized, the quACK expresses the equivalent of TCP's cumulative + selective ACK over opaque (randomly identified) packets in 48 bytes, tolerating up to 10 missing packets before the last "selective ACK." On a recent x86-64 CPU, it takes 33 ns/packet for a sidekick PEP to encode a quACK, and 3 μs for an end host to decode it. These overheads compared well with several alternatives (Section 3.5).

We implemented Robin in a low-latency media client based on the WebRTC standard, and an HTTP/3 client using the Cloudflare implementation of QUIC [13] and the `libcurl` [45] implementation of HTTP/3. We evaluated the three scenarios in real-world and emulation experiments. In real-world experiments using an unmodified local Wi-Fi network to access our nearest AWS datacenter, the sidekick was able to trigger early retransmissions to fill in gaps in the audio of a latency-sensitive audio stream, reducing the receiver's de-jitter delay from 2.3 seconds to 204 ms—about a 91% reduction (Figure 8). The sidekick was also able to improve the speed of an HTTP/3 (QUIC) upload by about 50%.

In emulation experiments of the "battery-powered receiver" scenario, the sidekick PEP was able to reduce the need for the receiver to send ACKs by sending proxy acknowledgments on its behalf—ACKs the sender used to advance its flow-control and congestion-control windows. The receiver only needed to wake up its radio to send occasional end-to-end ACKs, which the sender used to discard data from its buffer (Figure 4c).

Also in an emulation experiment, we confirmed that PACUBIC's performance approximates a split CUBIC connection (two TCP CUBIC connections separated by a PEP), responding to loss events on the different path segments similarly to how the individual CUBIC flows would (Figure 6). The results indicate that the sidekick protocol's gains do not come at the expense of congestion-control fairness relative to a split CUBIC connection.

The rest of this paper describes the sidekick's motivating scenarios (Section 2), explores the quACK's design and implementation (Section 3), discusses the concrete sidekick protocol we built around quACKs (Section 4) and its implementation in two base protocols (Section 5), and then evaluates the protocol in real-world and emulation experiments (Section 6).

2 Motivating Scenarios

We focus on three scenarios where end hosts benefit from in-network assistance. In each one, a proxy server provides feedback, called a quACK, to an end host: the data sender (Figure 1). Recall that a quACK is a "cumulative ACK + selective ACK" over encrypted sequence numbers. The data sender uses this feedback to influence its behavior on the base connection, without altering the wire format.

To be clear: the sidekick protocol is not tied to a specific base protocol nor to how the end hosts use the quACK infor-

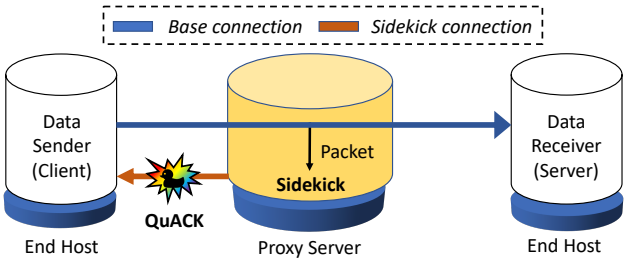


Figure 1: The proxy generates quACKs, in-network acknowledgments, based on the opaque packets it observes in the base protocol. It quACKs to an end host, the data sender, which sends or resends packets on the base protocol as a result. Although we only show one side of the connection, the sidekick could assist either end host of a bidirectional flow.

mation. The base protocol does not need to be reliable, nor to have unique datagrams—we implemented and evaluated the same sidekick protocol and the same middlebox behavior across the different scenarios in this paper.

2.1 Low-Latency Media

Consider a train passenger using on-board Wi-Fi to have a low-latency audio conversation, using WebRTC/SRTP [54], with a friend. The end-to-end network path contains a low-latency, high-loss "near" path segment (the Wi-Fi hop) followed by a high-latency, low-loss "far" path segment (the cellular and wired path over the Internet). The friend probably suffers from poor connection quality, experiencing drops in the audio stream or high de-jitter buffer delays from waiting for retransmitted packets to be played in order (Figure 4a in emulation, Figure 8a in real world).

In the sidekick approach, a sidekick on the Wi-Fi access point sends quACKs to the audio application on the user's laptop, assisting the base connection's data sender. The sender uses quACKs to retransmit packets sooner than they would have using negative acknowledgments (NACKs) from the receiver. The end result is similar to the effect of prior PEPs, such as Snoop [2] and Milliproxy [55], that leverage TCP's cleartext sequence numbers to trigger early retransmission on lossy wireless paths.

2.2 Connection-Splitting PEP Emulation

Consider the same train passenger as before but uploading a large file over the Internet with a reliable transport protocol. If the protocol were TCP, the train could deploy a split TCP PEP at the access point. The split connection allows quick detection and retransmission of dropped packets on the lossy Wi-Fi segment, while opening up the congestion window on the high-latency cellular segment.

However, opaque transport protocols like QUIC can't benefit from (nor be harmed by) connection-splitting PEPs. Without a PEP, QUIC relies on end-to-end mechanisms over the entire path to detect losses, recover from them, and adjust the congestion-control behavior. This leads to reduced upload speeds (Figure 4b in emulation, Figure 8b in real world).

With help from the same sidekick PEP, the QUIC sender combines information from quACKs and end-to-end ACKs to emulate the congestion-control behavior of a split TCP connection (Section 4.3.2). The application considers whether packets are lost on the near or far path segments, and adjusts the congestion window accordingly while respecting the opacity of the end-to-end base connection. The application also retransmits the packet as soon as the loss has been detected.

The only guarantee the proxy makes to the sender via the quACK is that it has received some packets. To respect the end-to-end reliability contract with the receiver, the sender does not delete packets that may need to be transmitted until it receives an ACK, even if the packet has been quACKed.

2.3 ACK Reduction

Now consider a battery-powered device downloading a large file from the Internet. To reduce how often the receiver's radio needs to wake up, saving energy, the base connection can reduce the frequency of end-to-end ACKs the device sends. ACK reduction has also been shown to improve performance by reducing collisions and contention over half-duplex links [16, 43]. The ACK frequency can be configured with a TCP kernel setting or proposed QUIC extension [31].

However, ACK reduction can also degrade throughput [15, 16] (Figure 4c in emulation). The sender receives more delayed feedback about loss, and has to carefully pace packets to avoid bursts in the large delay between ACKs. One proposal has the PEP acknowledge packets on behalf of the receiver [36], leveraging cleartext TCP sequence numbers, but it does not apply to opaque transport protocols.

In this case, a sidekick at the Wi-Fi access point (or a cellular base station) quACKs to the sender on behalf of the receiver. The receiver still occasionally wakes up its radio to send ACKs, but the sender uses the more frequent quACKs to advance its flow-control and congestion-control windows.

The sender respects the end-to-end reliability contract by only deleting packets in response to ACKs, but disregards the receiver's flow control by using quACKs to advance the flow-control window. If the sender only used ACKs to advance the window, it would waste time waiting between ACKs to send packets with too small a window, and need to pace sent packets on receiving a large ACK with too large a window.

3 QuACK

As previously illustrated, a sidekick needs to be able to refer to and efficiently acknowledge a set of opaque packets seen by a network intermediary. But this problem is technically challenging for middleboxes without access to cleartext sequence numbers or the ossification of other fields.

We start by mathematically defining the quACK problem. We discuss how to select an *identifier* to refer to a packet, and analyze strawman solutions to the quACK problem that

use too much space or computation. Finally, we present an efficient construction of a quACK based on the insight that we can model the problem as a system of power sum polynomial equations when we have a bound on the maximum number of missing elements, a threshold t. This solution is most similar to the deterministic solution to the *straggler identification* problem [22], and also builds on related theoretical work in set reconciliation [50], and coding theory and graph theory [35].

3.1 The QuACK Problem

We first describe the quACK problem. A data sender transmits a multiset[1] of elements S (these correspond to packets). At any given time, a receiver (such as a proxy server) 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—the set difference $S \setminus R$—knowing S. We call this small amount of information the "quACK", and the problem is: **what is in a quACK and how do we decode it?**

3.2 Packet Identifiers

In a networking context, how exactly do we refer to the elements in the quACK problem that have been sent or received? Traditional TCP middleboxes have been able to interpose their own concise, cumulative acknowledgments using cleartext sequence numbers, but this is not possible with modern, secure transport protocols. Even if a connection did expose an unencrypted numerical field, we would not want to refer to that field at risk of ossifying that protocol.

Instead, we need a function that deterministically maps a packet to a random b-byte *identifier*. The most trivial solution that applies to all base protocols is to hash the entire payload. Another option if the payload is already pseudorandom (e.g., QUIC) is to take the first b bytes from a fixed offset of that payload. Although the latter option would rely on those bytes to remain pseudorandom, it is computationally more efficient because it does not require reading the entire payload.

Collisions. The main considerations when selecting the number of bytes, b, in an identifier is the tolerance for collisions compared with the extra data needed to refer to these packets on the link. The larger b is, the lower the collision probability but the greater the link overhead.

Define the collision probability to be the probability that a randomly-chosen b-byte identifier in a list of n packets maps to more than one packet in that list. If we assume that identifiers are uniformly distributed, this probability is equal to $1-(1-1/256^b)^{n-1}$. When $n=25$, using 4 bytes results in an almost negligible chance of collision while using 2 bytes results in a 0.04% chance (Table 1).

When handling collisions, a sender who is decoding a quACK has a list of n packets it is trying to classify as received or missing (Section 3.5). Note that collisions are also known to the sender beforehand. If there is a collision between a packet that is received and a packet that is missing, the fate of that identifier is considered indeterminate. In our scenarios (Section 2), either the protocol can still function with approximate statistics (e.g., congestion control) or it can fall back to an end-to-end mechanism (e.g., retransmission).

[1] A "multiset" means the same element can be transmitted more than once.

Identifier Bytes	1	2	4	8
Collision Prob.	0.090	0.0004	5.6e-09	≈0

Table 1: Collision probabilities for $n = 25$.

3.3 Strawman Solutions

A problem that is simple with cumulative and selective acknowledgments of plaintext sequence numbers is deceivingly challenging for pseudorandom packet identifiers. Consider the following strawman solutions to the quACK problem:

Strawman 1: Echo every identifier. Strawman 1a, similar to [41, 44], echoes the identifier of every received packet in a new UDP packet to the data sender. Decoding is trivial given the identifiers are unmodified. This strawman adds significant link overhead in terms of additional packets. Additionally, since the strawman is not cumulative, losing a quACK means the end host could falsely consider a packet to be lost, creating a congestion event or spurious retransmission.

Strawman 1b echoes a sliding window of identifiers over UDP such that there is overlap in the identifiers referred to by consecutive quACKs. This solution is slightly more resilient to loss, but uses more bytes and is still not guaranteed to be reliable. Another variant batches identifiers to reduce the number of packets, but this solution is even less resilient to loss.

We also consider a Strawman 1c that echoes every identifier over TCP with `TCP_NODELAY` to send every identifier in its own packet. This ensures there are no false positives when detecting lost packets, but adds even more link overhead in terms of TCP headers and additional ACKs from the data sender (every other packet by default in the Linux kernel).

Strawman 2: Cumulative hash of every identifier. Strawman 2 sends a SHA-256 hash of a sorted concatenation of all the received packets in a UDP packet, and the sender hashes every subset of the same size of sent packets until it finds the subset with the same hash (assuming collision resistance). The strawman includes a count of the packets received to determine the size of the subset to hash. As the number of missing packets exceeds even a moderate amount, the number of subsets to calculate explodes, making the strawman impractical to decode.

One might also suggest the receiver send negative acknowledgments of the packets it has not received. However, unlike sequence numbers where one can determine a gap in received packets, there is no way to tell with random identifiers what packet is missing or should be expected next.

	Per-Packet Encode Time	Decode Time	Num Additional Proxy Packets	Packet Payload Size (bytes)	Cumulative?
Strawman 1a	Parse identifier	N/A	n	b	No
Strawman 1b	Parse identifier, move sliding window	N/A	n	$b \cdot window$	No
Strawman 1c	Parse identifier	N/A	n (TCP headers)	b	No
Strawman 2	Parse identifier, concatenate and hash	Concatenate and hash $\binom{n}{m}$ subsets	1	$32+4$ (hash and count)	Yes
Power Sums	Parse identifier, t modular multiplications and additions	Plug n candidate roots into a degree-m polynomial OR solve system of m polynomial equations	1	$4+b+b \cdot t$ (count, last value, t power sums)	Yes

Table 2: Strawmen compared to the power sum quACK representing n packets sent by the data sender, m missing packets, and b-byte identifiers. The power sum quACK uses the threshold t. The total data overhead of each quACK must consider the packet payload size along with transport headers. We evaluate the overheads in practice in Section 6.4.

3.4 The Power Sum Solution

Now we describe a solution to the quACK problem based on the insight that we can model the problem as a system of power sum polynomial equations when we have a bound on the maximum number of missing elements, a threshold t. Unlike the previous strawmen, this construction is efficient to decode, and its size is proportional only to t.

Consider the simplest case, when the receiver is only missing a single element. The receiver maps packet identifiers to a finite field, i.e. modulo the largest prime that fits in b bytes, and communicates 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 m equations:

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

Instead of transmitting an unbounded number of power sums, the receiver only maintains and sends the first t power sums. Efficiently solving these t power sum polynomial equations in t variables in a finite field is a well-understood algebra problem [22]. The solutions are exactly $x \in S \setminus R$.

Efficiency. The power sum quACK is efficient to decode, adds reasonable link overhead, and is a cumulative representation of the packets seen by the receiver (Table 2). Compared to Strawman 2, the power sum quACK can be decoded with simple algebraic techniques. Its link overhead is proportional only to the number of missing packets between consecutive quACKs, up to a configurable threshold. In comparison, the link overhead of Strawman 1 is necessarily proportional to the number of received packets. The power sum quACK is also resilient to mis-identifying a received packet as dropped, in the case a quACK is lost in transmission.

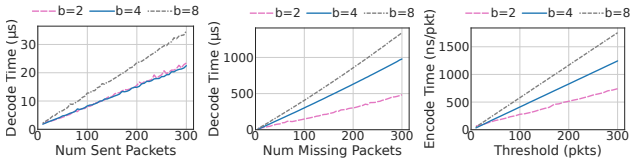
Interface. The actual format of the power sum quACK includes three fields: (i) t b-byte power sums, (ii) a 4-byte count of received elements, and (iii) the b-byte identifier of the last element received. We assume power sum quACKs to be sent over UDP, though the actual mechanism is not tied to the design. Since the decoder does not know m ahead of time, the decoder takes the difference between the number of packets it has sent and the count in the quACK to calculate m. Sending the last element received is an optimization that allows m to represent just the "holes" among the packets being selectively ACKed, excluding the possibly many consecutive elements that are in-flight (Section 4.3.1).

3.5 Microbenchmarks

We benchmark our optimized implementation of the power sum quACK [65] to demonstrate its practicality for in-line packet processing. Our microbenchmarks used an m4.xlarge AWS instance with a 4-CPU Intel Xeon E5 processor @ 2.30 GHz and 16 GB memory.

	Encode Time	Decode Time
Strawman 1a/1c	1 ns/pkt	0
Strawman 1b	51 ns/pkt	0
Strawman 2	27 ns/pkt	830 ms
Power Sum	**33 ns/pkt**	**2.82 μs**

Table 3: The CPU overheads of power sums are comparable to those of the strawmen, while being more efficient in space and computation. The encode time includes constructing and serializing the quACK(s), given n identifiers. The decode time includes finding the identifiers of either R or $S \setminus R$, given the quACK(s) and S. Parameters: $n = 25$, $t = 10$, $b = 4$.



(a) Evaluate a degree-$m = t = 10$ polynomial at n candidate roots.

(b) Plug $n = 300$ candidate roots into a degree-m polynomial.

(c) Update t power sum equations. Average of 1000 packets.

Figure 2: How power sum quACK performance depends on various parameters: bit width, threshold, number of sent and missing packets. Average of 100 trials.

A power sum quACK that represents $n = 25$ outstanding packets (packets in consideration on that path segment not yet known to be received or lost) and up to $t = 10$ missing packets with $b = 4$-byte identifiers adds 33 ns of encoding time per packet and takes 2.82 μs to decode (Table 3).

Decoding. The decode time must be comparable to the time it takes to process a typical ACK and modify the logic in the transport protocol. Decoding typically occurs on end hosts, compared to encoding which occurs in the middle of the path.

Finding the solution to the system of power sum polynomial equations boils down to applying Newton's identities (a linear algorithm) and finding the roots of a polynomial equation in a modular field [22]. Factoring a polynomial is asymptotically fast in theory, but the implementation is branch-heavy and complicated [3]. We found that plugging in and evaluating which of n candidate roots evaluated to zero was faster in practice for $n < 40{,}000$ roots. This is the method we use to decode the power sum quACK.

The decode time of this method is directly proportional to n (Figure 2a) and the number of missing packets m (Figure 2b). Decoding takes $2820/10/25 \approx 11$ ns/candidate/missing. Both n and m are typically a few hundred at most.

Encoding. The encode time per-packet is directly proportional to the threshold number of missing packets t (Figure 2c) at $33/10 \approx 3$ ns/power sum. Each power sum can be updated in a constant number of operations based on the previous power sum, so encoding an identifier requires t modular additions and multiplications for the t power sums.

Bit widths. Different bit widths have different implications for which instructions the CPU can use. Modular operations are efficient for 16- and 32-bit integers, fitting within the 64-bit word size (the number of bits that can be processed in one instruction) of most modern CPUs. For example, to multiply two 32-bit integers, we cast them to 64-bit integers, multiply, then take the modulus.

Figure 2 shows the best performance we achieved at different bit widths. For 16-bit identifiers only, we precomputed power tables that fit in the L3 cache. For 64-bit identifiers, we implemented Montgomery modular multiplication [51] to avoid an expensive hardware division for 128-bit integers. In the remainder of the paper, we use $b = 4$ as the preferred tradeoff between space and collision probability.

4 Sidekick Protocol

This section describes Robin, our design for a sidekick protocol built around quACKs. This includes the setup and configuration of a Robin sidekick connection, how a sender detects loss from a quACK, and a path-aware modification to CUBIC called PACUBIC, for congestion-controlled base protocols.

4.1 PEP Discovery Mechanism

Sidekick connections can be configured explicitly or implicitly. In systems that explicitly configure proxies, such as Apple's iCloud Private Relay [1] based on MASQUE [39, 40], proxies can simply negotiate sending quACKs during session establishment. In most other settings, such as 4G/5G cellular networks, PEPs have traditionally been deployed as transparent proxies, silently interposing on end-to-end connections. Senders therefore need a way to detect transparent sidekick proxies and inform them of where to send quACKs. Because of network address translation, all communication to the proxy must be initiated by the sender or use the same IP addresses and port numbers of the base connection.

Our current design has senders signal quACK support by sending a distinguished packet containing a 128-byte *sidekick-request* marker. Such inline signaling could confuse receivers, but sidekicks target protocols such as QUIC that discard cryptographically unauthenticated data anyway. It would be cleaner to signal support through out-of-band UDP options [61], which we hope to do once they are standardized.

The proxy replies to a sidekick-request packet by sending a special packet from the receiver's IP address and port number back to the sender. This packet contains a *sidekick-reply* marker, an opaque session ID, and an IP address and port number for communicating with the proxy. Upon receiving the sidekick-reply packet, the sender begins communicating directly with the proxy from a different UDP port. It initially sends back the session ID and configuration parameters to start receiving quACKs.

Security. A malicious third-party could execute a reflection amplification attack that generates a large amount of traffic while hiding its source. This is possible because the sender requests quACKs to a different port and (for some carrier-grade NATs) IP address from the underlying session. To mitigate this, each quACK contains a quota, initially 1, of remaining quACKs the proxy will send as well as an updated session ID. The quota and session ID ensure only the sender can increase the quota or otherwise reconfigure the session.

An adversarial PEP could send misleading information to the sender. Note that only on-path PEPs can send credible information, since they refer to unique packet identifiers. To mitigate this, the sender can consider PEP feedback along with end-to-end metrics to determine whether to keep using the PEP. The sender can always opt out of the PEP, and the

PEP cannot actively manipulate traffic any more than outside a sidekick setting.

4.2 Configuration Messages

The data sender can send various other messages to the proxy to configure the connection or reset bad state.

Protocol parameters. The sender configures (i) the quACK interval of the PEP and (ii) the threshold number of missing packets t, or otherwise selects sidekick-specific settings such as how an identifier is computed.

The quACK interval is expressed in terms of time or number of packets, e.g., every N milliseconds or every N packets, as in a TCP delayed ACK. The sender determines the desired interval based on its estimated RTT of the base connection and its application objectives, e.g., more frequently for latency-sensitive applications or lower-RTT paths.

The threshold represents the bound on the number of missing packets between quACKs, in practice the number of "holes" among the packets that are selectively ACKed. The threshold depends on the quACK interval, and should be set based on how precise loss detection needs to be and other qualities of the link. For example, the threshold is larger to detect congestive loss in the queue of a bottleneck link, or smaller to still detect transmission error on a lossy link.

Resets. Robin allows the sender to tell the PEP to reinitialize the quACK. This is helpful if the quACK becomes invalid, e.g., if m exceeds the threshold t. It is always safe to reset the quACK, or even to ignore the sidekick entirely and fall back to the base protocol's end-to-end mechanisms.

4.3 Sender Behavior

In this section, we discuss two particular sender-side behaviors that are enabled by the sidekick protocol and which are helpful across several scenarios: detecting packet loss from a decoded quACK and congestion control.

4.3.1 Detecting Loss

The sender knows definitively which packets have been received by the proxy from a decoded quACK. Next, it must determine from the remaining packets which ones have been dropped and which are still in-flight, including if there has been a reordering of packets. In-flight packets are later classified as received or dropped based on future quACKs.

When there is no reordering, the packets that are dropped are just the "holes" among the packets that are selectively ACKed by the quACK. In particular, these are the holes when considering sent packets in the order they were sent up to the last element received, which represents the last selective ACK. To identify these dropped packets, the sender encodes t cumulative power sums of its sent packets up to the last element received. The difference between these power sums and the power sums in the quACK represents the dropped packets. The sender "removes" the identifiers of dropped packets from its cumulative power sums, ensuring that the only packets that contribute to the threshold limit are those that went missing since decoding the last quACK.

To account for reordering in loss detection, Robin implements an algorithm similar to the 3-duplicate ACK rule in TCP [5, 60]. In TCP, if three or more duplicate ACKs are received in a row, it is a strong indication that a segment has been lost. Robin considers a packet lost only if three or more packets sent after the missing packet have been received. Other mechanisms could involve timeouts for individual packets similar to the RACK-TLP loss detection algorithm for TCP [11].

4.3.2 Path-Aware CUBIC Congestion Control

Congestion-controlled base protocols must have a congestion response to lost packets that they retransmit due to quACKs, similar to if the loss were discovered by the end-to-end ACK. This ensures friendliness with end-to-end congestion control algorithms that do consider the loss, such as CUBIC [27] in the presence of a connection-splitting TCP PEP. Here, we propose PACUBIC, an algorithm that emulates this "split CUBIC" behavior. PACUBIC uses knowledge of where loss occurs to improve connection throughput compared to end-to-end CUBIC, while remaining fair to competing flows.

Recall that CUBIC [27] reduces its congestion window by a multiplicative decrease factor, $\beta = \beta^* = 0.7$, when observing loss (a congestion event), and otherwise increases its window based on a real-time dependent cubic function with scaling factor $C = C^* = 0.4$:

$$cwnd = C(T-K)^3 + w_{max} \text{ where } K = \sqrt[3]{\frac{w_{max}(1-\beta)}{C}}.$$

Here, $cwnd$ is the current congestion window, w_{max} is the window size just before the last reduction, and T is the time elapsed since the last window reduction.

While a split CUBIC connection has *two* congestion windows, end-to-end PACUBIC only has *one* window representing the in-flight bytes of the end-to-end connection. Conceptually, we want an algorithm that enables PACUBIC's single congestion window to match the sum of the split connection's two congestion windows.

PACUBIC effectively makes it so that we reduce and grow $cwnd$ proportionally to the number of in-flight bytes on the path segment of where the last congestion event occurred. Let r be the estimated ratio of the RTT of the near path segment (between the data sender and the proxy) to the RTT of the entire connection (between end hosts). We use r as a proxy for the ratio of the number of in-flight bytes. If the last congestion event came from a quACK, we use the same real-time dependent cubic function but with the following constants[2]

$$\beta = 1 - r(1-\beta^*) \text{ and } C = \frac{C^*}{r^3}.$$

[2]See Appendix A for more intuition behind β' and C'.

If the last congestion event came from an end-to-end ACK, then we use the original β and C as above.

While this algorithm resembles the congestion behavior of split CUBIC, it is simply an approximation. PACUBIC does not know the exact number of bytes in-flight on each path segment, and the sum of the two congestion windows is simply a heuristic for an inherently different split connection. The main takeaway is that knowing where loss occurs can inform congestion control. We generally hope that quACKs can lead to the development of smarter, path-aware algorithms.

5 Implementation

Module	Language	LOC
QuACK library (Section 3.5)	Rust	1772
Media server/client + integration	Rust	478
`quiche` client integration	Rust	1821
`libcurl` client integration	C	1459
Proxy sidekick binary	Rust	833

Table 4: Lines of code.

We now describe our implementation of Robin [66] for several applications. We integrated sidekick functionality with a simple media client for low-latency streaming and an HTTP/3 (QUIC) client. The total implementation of the quACK library, and proxy and client integrations used 6363 LOC (Table 4).

5.1 Baselines and Applications

The baselines we evaluated against were the performance of two opaque transport protocols without proxy assistance, and the fairness of a split CUBIC connection.

Low-latency media application. We implemented a simple server and client in Rust for streaming low-latency media. The client sends a numbered packet containing 240 bytes of data every 20 milliseconds, representing an audio stream at 96 kbit/s. The sequence number is encrypted on the wire.

The server receives packets. If it receives a nonconsecutive sequence number, it sends a NACK back to the client that contains the sequence number of each missing packet. The client's behavior on NACK is to retransmit the packet. The server retransmits NACKs, up to one per RTT, until it has received the packet.

The server's application behavior is to store incoming packets in a buffer and play them as soon as the next packet in the sequence is available. The de-jitter buffer delay is the length of time between when the packet is stored to when it can be played in-order. Some packets can be played immediately.

HTTP/3 file upload application. We used the popular `libcurl` [45] file transfer library as the basis for our HTTP client, and an `nginx` webserver. The client makes an HTTP POST request to the server. Both are patched with `quiche` [13], a production implementation of the QUIC protocol from Cloudflare, to provide support for HTTP/3.

For our TCP baselines, we used the same file upload application with the default HTTP/1.1 server and client. We used a split-connection TCP PEP [10] that intercepts the TCP SYN packet in the three-way handshake, pretends to be the other side of that connection, and initiates a new connection to the real endpoint. Both clients use CUBIC congestion control.

5.2 Client Integration

In each application, we modified only the *client* to speak Robin and respond to in-network feedback. The server remained unchanged. The modifications were in two parts: following the discovery mechanism to establish bi-directional communication with the proxy, and using the information in the quACK to modify transport layer behavior.

Low-latency media client. The media client has two open UDP sockets: one for the base connection and one for the sidekick connection. When it receives a quACK, it detects lost packets without reordering and immediately retransmits them. The protocol does not have a congestion window nor a flow-control window. The client also sends reset and configuration messages over the sidekick connection.

HTTP/3 file upload client. The HTTP/3 client similarly has an adjacent UDP socket for the sidekick connection on which it receives quACKs and sends reset and configuration messages. The client passes the quACK to our modified `quiche` library, which interprets the quACK and makes transport layer decisions. From the client's perspective, `quiche` tells `libcurl` exactly what bytes to send over the wire.

Our modified `quiche` library uses the quACK to inform the retransmission behavior, congestion window, and flow-control window. The library immediately retransmits lost *frames* in a newly-numbered packet, as opposed to the lost *packet*, similar to QUIC's original retransmission mechanism. We implement PACUBIC, described in Section 4.3.2. We also move the flow-control window (without forgetting packets in the retransmission buffer), but only in the ACK reduction scenario, when the congestion window is nearly representative of that of the sidekick connection's path segment.

5.3 Proxy Integration

Our proxy sniffs incoming packets of a network interface using the `recvfrom` system call on a raw socket. It stores a hash table using Rust's standard library `HashMap` that maps socket pairs to their respective quACKs, and incrementally updates the quACKs for flows that have requested sidekick assistance. It also sends quACKs at their configured frequencies and listens for configuration messages.

6 Evaluation

We evaluated Robin to answer the following questions:

1. Can sidekicks improve the performance of opaque transport protocols in a variety of scenarios while preserving the opaque behavior of the base protocols?

Scenario	Link 1	Link 2	QuACK Interval	Threshold	Success Metric	Emulated?	Real-World?
#1 Low-latency media	1 ms delay, 3.6% loss, 100 Mbit/s	25 ms delay, 0% loss, 10 Mbit/s	2 pkts	8	Reduce tail latency of how long packets are queued in the data receiver's de-jitter buffer.	Yes	Yes
#2 Connection- splitting PEP emulation	1 ms delay, 1.0% loss, 100 Mbit/s	25 ms delay, 0% loss, 10 Mbit/s	30 ms	10	Achieve high throughput; match the performance, congestion control behavior, and fairness of connection-splitting TCP PEPs.	Yes	Yes
#3 ACK reduction	25 ms delay, 0% loss, 10 Mbit/s	1 ms delay, 0% loss, 100 Mbit/s	15 ms	50	Reduce ACK frequency of data receiver; achieve high throughput.	Yes	No

Table 5: Experimental scenarios. Link 1 connects the data sender (client) to the proxy, while Link 2 connects the proxy to the data receiver (server). The quACK interval and threshold represent our sidekick configuration.

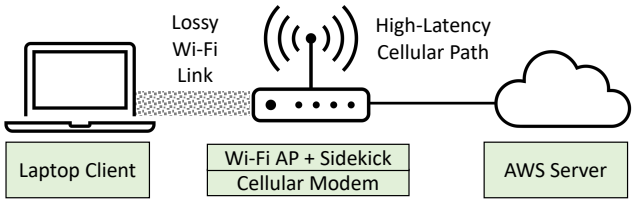


Figure 3: Real-world experimental setup.

2. Can a path-aware congestion control algorithm match the fairness of split TCP PEPs using CUBIC?
3. How do the CPU overheads of encoding quACKs impact the maximum capacity of a proxy with a sidekick?
4. What link overheads does the power sum quACK add and how does it compare to the strawmen?
5. Is Robin robust in a real-world environment?

6.1 Experimental Setup

We modeled the scenarios from Section 2 in both emulated and real-world environments. We answer questions 1-4 in emulation and question 5 in the real world. We use the same m4.xlarge AWS instance as before for the emulated experiments, and as the server in the real-world experiments.

Emulated environment. We emulated a two-hop network topology (Figure 1) in mininet, configuring the link properties using `tc`. In emulation, we represented each link by a constant delay (with variability induced by the queue), a random loss percentage, and a maximum bandwidth. Table 5 describes the parameters for each link to model—e.g., lossy Wi-Fi or a high-latency cellular path—as well as the metrics for success in that scenario. Link 1 connects the data sender (client) to the proxy, while Link 2 connects the proxy to the data receiver (server). On the proxy, we either run a sidekick, a connection-splitting TCP PEP [10], or nothing at all.

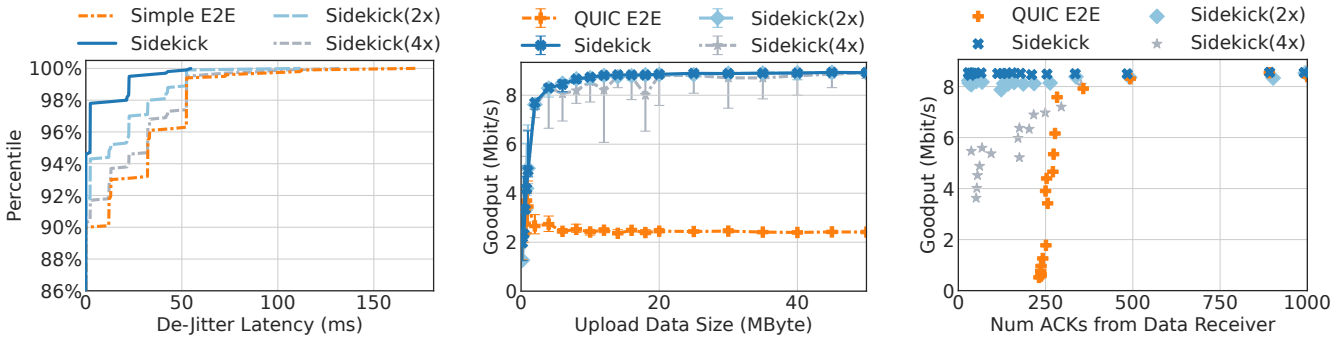
Real-world environment. To test its robustness, we also evaluated Robin over a real-world environment that resembled the scenario on the train (Figure 3). In this setup, a Lenovo ThinkPad laptop, running Ubuntu 22.04.3 with a 4-Core Intel i7 CPU @ 2.60 GHz and 16 GB memory, acted as a client to an AWS instance in the nearest geographical region. The ThinkPad used as an access point (AP) a Lenovo Yoga laptop, running Ubuntu 20.04.6 with a 4-Core Intel i5 CPU @ 1.60 GHz and 4 GB memory, with a 2.4 GHz Wi-Fi hotspot. The AP was connected to the Internet via a JEXtream cellular modem with a 5G data plan. The AP ran sidekick software.

We measured the link properties of each path segment to compare to our emulation parameters. We measured delay and loss using 1000 `pings` over a 100 second period, and bandwidth using an `iperf3` test. On the near segment between the ThinkPad client and the AP, the min/avg/max/stdev RTT was 1.249/37.194/272.168/54.660 ms at 49.8 Mbit/s bandwidth. We observed that loss increased the further away the AP. In our experiments, the client was located roughly 200 feet away in a different room, with 3.6% loss. The far segment between the AP and the AWS server was 48.546/64.381/92.374/6.806 ms with 0.0% loss at 30.9 Mbit/s. In both environments, the cellular link was the bottleneck link in terms of bandwidth, and the corresponding path segments in emulation had similar minimum RTTs and average loss percentages.

6.2 Performance Comparison to Baseline

We first evaluate Robin's main performance goal: In each of the motivating scenarios, we show that Robin can improve performance compared to the base protocol alone, which would not be able to benefit from existing PEPs. Each scenario has a different metric for success—tail latency, throughput, or number of packets sent by the data receiver (corresponding to energy usage or chance of Wi-Fi collisions)—demonstrating the versatility of the sidekick protocol.

Low-Latency Media. The sidekick can reduce tail latencies in a low-latency media stream, representing fewer drops and better quality of experience. The early retransmissions induced by the sidekick reduced the 99th percentile latency of the de-jitter buffer delay from 48.6 ms to 2.2 ms—a 95% reduction (Figure 4a). As long as the quACK interval is less than the end-to-end RTT, the connection benefits from the sidekick.



(a) Scenario #1: Low-latency media. Reduced tail latency of de-jitter delay with earlier retransmission. 5 minute trials.

(b) Scenario #2: Connection-splitting PEP emulation. Improved goodput. 20 trials median. Error bars are 1st and 3rd quartiles.

(c) Scenario #3: ACK reduction. High goodput independent of end-to-end ACK frequency. 10 MB upload.

Figure 4: Comparing the end-to-end baseline protocol to the same protocol with a sidekick connection, using the success metrics for the three scenarios described in Table 5. The Sidekick (Nx) data points show the performance at Nx the quACK interval (sent less frequently) and threshold of the default configurations specified in Table 5.

The sidekick is beneficial in this scenario because it enables the client to sooner detect and retransmit lost packets, and the server to sooner play packets from its de-jitter buffer. The end-to-end mechanism takes one additional received packet to notify of the loss and one end-to-end RTT to retransmit and play the packet (20+52=72ms), resulting in three delayed packets (the three "steps" in Figure 4a) in most cases. The sidekick takes up to two additional packets and one near path segment RTT (20+2=22ms or 20×2+2=42ms), delaying either one or two packets in comparison. Dropped ACKs and quACKs account for the $< 2\%$ of packets with even greater de-jitter latencies.

Connection-Splitting PEP Emulation. The sidekick improves upload speeds when there is a lossy, low-latency link by using quACKs to inform the sender's congestion control. In a scenario with 1% random loss on the link between the proxy and the data sender, the HTTP/3 (QUIC) client achieves 3.6× the goodput for a 10 MB upload with a sidekick compared to end-to-end QUIC (Figure 4b).

When there is no random loss, the sidekick does not impact the performance of QUIC. There are no logical changes to the base protocol in this case because all loss is on the bottleneck link on the far path segment, and the CPU overheads of processing quACKs are negligible.

Knowing *where* congestion occurs is an opportunity for creating smarter congestion control. In PACUBIC, identifying where the loss occured let the data sender reduce the congestion window proportionally to how many packets were in-flight on each path segment. In Section 6.3, we will show that our path-aware congestion control algorithm still matches the fairness of connection-splitting TCP PEPs.

ACK Reduction. Using quACKs in lieu of end-to-end ACKs allows the data receiver to significantly reduce its ACK frequency while maintaining high goodput. In our experiment, QUIC with a sidekick sent 96% fewer packets (mainly ACKs) than end-to-end QUIC before the goodput dropped below 8.5 Mbit/s (Figure 4c). The quACK enables the data sender to promptly move the flow-control window forward, as long as the last hop is reliable.

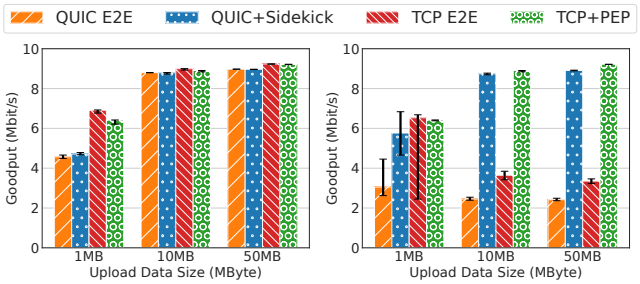
The goodput significantly degrades when reducing the end-to-end ACK frequency without a sidekick. When end-to-end QUIC reduces the ACK frequency to every 80 ms, the data receiver sends $247/138 = 1.8\times$ the packets at $4.5/8.4 = 0.5\times$ the goodput, worse than QUIC with the sidekick in both dimensions (Figure 4c). With a sidekick, the data sender also does not need to change packet pacing to avoid bursts in response to infrequent ACKs, which is why end-to-end QUIC cannot send fewer than ≈ 240 packets.

6.2.1 Configuring the Sidekick Connection

Table 5 shows the quACK interval and threshold we elected for each scenario based on the considerations in Section 4.2. In each experiment in Figure 4, we also show how with less frequent quACKs (2× and 4× the interval) and proportionally-adjusted thresholds, the protocol performs worse, or more variably. Less frequent quACKs means the client reacts later to feedback about the near path segment, and more often has to rely on the end-to-end mechanism. The performance particularly degrades when the quACK interval exceeds the end-to-end RTT. However, even in this case, the base protocol with any sidekick at all performs better than the base protocol alone.

6.3 Fairness Evaluation

It is easy to improve performance without regard to competing flows; however, we demonstrate that PACUBIC can match the fairness of split CUBIC in a TCP PEP connection. We evaluate fairness using Scenario #2 with varying amounts of loss on the near path segment.



(a) 0% loss. (b) 1% loss.

Figure 5: Median goodput for three upload data sizes with 0% and 1% loss on Link 1. 20 trials. Error bars are 1st and 3rd quartiles. With proxy assistance at 1% loss, both QUIC and TCP match the performance of when there is no loss at all.

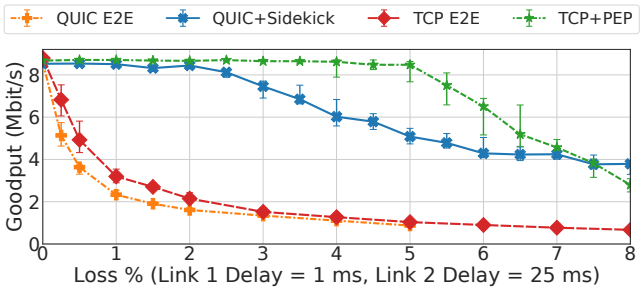


Figure 6: Connection-splitting PEP emulation as a function of near-segment loss rate. In this emulation experiment, QUIC+Sidekick (running PACUBIC) performs similarly to TCP+PEP (each connection running CUBIC) and improves goodput compared with end-to-end protocols. The graph shows median goodput of a 10 MByte upload. QuACK interval is 30 ms, threshold is 10. Error bars show IQR of 10 trials.

QUIC vs. TCP. We first compare QUIC to TCP without either PEP. As both connections use CUBIC, they exhibit similar congestion control behavior and achieve nearly maximum throughput in the emulated network with no random loss (Figure 5a). We attribute the differences to the slightly different retranmission and loss recovery behaviors of QUIC and TCP. The PEPs do not affect the performance.

With even a little loss on the near path segment, both QUIC and TCP dramatically worsen, respectively achieving 28% and 42% of the goodput at 0% loss, for a 10 MB upload (Figure 5b). In both protocols, CUBIC treats every transmission error as a congestion event, even though no amount of reducing the congestion window affects the error rate. QUIC and TCP perform similarly to each other with proxy assistance and 1% loss on the near path segment.

Sidekick vs. TCP PEP. Figure 6 shows that QUIC with a sidekick roughly matches—as intended—the behavior of TCP with a PEP-assisted split connection. At higher loss rates, the near path segment becomes the bottleneck link even with earlier feedback about loss, causing the performance of TCP with proxy assistance to drop. QUIC with a sidekick follows a similar pattern because of its path-aware congestion-control scheme (Section 4.3.2). The results indicate that the sidekick protocol's gains do not come at the expense of congestion-control fairness relative to the split TCP connection.

6.4 Proxy CPU Overheads

	25-Byte Payload Cycles	%	1468-Byte Payload Cycles	%
Sniff Packet	22417	97.6	22408	97.5
Table Lookup	247	1.1	251	1.1
Parse ID	23	0.1	22	0.1
Encode ID	74	0.3	69	0.3
Other	213	0.9	225	1.0
Total	*22974*	*100.0*	*22975*	*100.0*

Table 6: Breakdown of the CPU cycles spent processing each packet at the proxy. Most cycles are spent on general per-packet overheads as opposed to quACK-specific processing.

The main bottleneck of Robin on a proxy is the CPU. Table 6 shows a breakdown of the number of CPU cycles in each step. The largest overhead was reading the packet contents from the network interface (97.5% of the CPU cycles).

Encoding an identifier in a power sum quACK with $t = 10$ used 74 CPU cycles (0.9%). As a calculation of the theoretical maximum on a 2.30 GHz CPU, the proxy would be able to process 31 million packets/second on a single core. The hash table lookup used 251 cycles and parsing the pseudorandom payload as an identifier used 22 cycles.

In practice, we measured the maximum throughput of Robin to be 464k packets/s with 25-byte payloads and 5.5 Gbit/s (458k packets/s) with 1468-byte packet payloads on a single core (assuming 1500-byte MTUs). This experiment used multiple `iperf3` clients to simulate high load until Robin was unable to keep up with the load on a single core. The packet payload size did not seem to affect results.

We find these achieved throughputs acceptable for edge routers such as Wi-Fi APs and base stations. To deploy Robin on core routers, we would need to reduce the overhead of reading packets from the NIC, such as by bypassing the kernel/user-space protection boundary[3] [20, 48, 62] or using native hardware [8]. We could also scale on multiple cores using symmetric RSS hashing [64].

6.5 Link Overheads

The other cost in terms of using sidekick protocols is the additional data sent by the proxy to the data sender. Too many

[3] A kernel-bypass system like Retina [62] can achieve 25 Gbps on 2 cores while processing raw packets with a 1000-cycle callback (Figure 5(a) in [62]). The Sidekick equivalent would be a 500-cycle callback, and assuming all traffic has requested sidekick help. Throughput scales almost linearly with the number of cores using symmetric RSS hashing. Thus we don't expect proxy overheads to be an issue with modern 100 Gbps network speeds and an optimized implementation even on commodity hardware.

	Data Sender→		←Proxy		←Data Receiver		
	Pkts	**Bytes**	**Pkts**	**Bytes**	**Pkts**	**Bytes**	**Goodput**
QUIC E2E	1.00×	1.00×	1.00×	1.00×	1.00×	1.00×	1.00×
Strawman 1a	0.96×	1.01×	2.02×	1.56×	1.01×	1.03×	3.33×
Strawman 1b	0.94×	1.00×	2.00×	1.78×	1.00×	1.03×	3.53×
Strawman 1c	1.83×	1.06×	2.01×	1.83×	1.00×	1.03×	3.46×
Power Sum	**0.94×**	**1.00×**	**1.03×**	**1.07×**	**1.00×**	**1.03×**	**3.55×**

(a) Scenario #2: Connection-splitting PEP emulation.

	Data Sender→		←Proxy		←Data Receiver		
	Pkts	**Bytes**	**Pkts**	**Bytes**	**Pkts**	**Bytes**	**Goodput**
QUIC E2E	1.00×	1.00×	1.00×	1.00×	1.00×	1.00×	1.00×
Strawman 1a	0.96×	1.00×	9.94×	4.99×	0.04×	0.08×	1.02×
Strawman 1b	0.96×	1.00×	9.95×	7.13×	0.04×	0.08×	1.02×
Strawman 1c	1.91×	1.05×	9.73×	7.41×	0.04×	0.08×	0.97×
Power Sum	**0.96×**	**1.00×**	**1.09×**	**2.56×**	**0.04×**	**0.08×**	**0.98×**

(b) Scenario #3: ACK reduction.

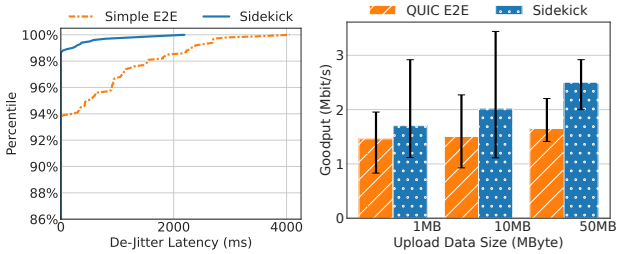
Figure 7: Link overheads for a 10 MB upload. The cells represent the multiplier relative to the end-to-end QUIC baseline for each type of quACK. Lower is better for number of packets and bytes sent on a link. Higher goodput is better. Robin's power sum quACK achieves the success metric for each scenario without incurring the link overheads of the strawmen. We did not evaluate the contrived protocol in Scenario #1.

additional bytes use up bandwidth, and additional packets use up CPU. Figure 7 shows the number of packets and bytes sent at each node comparing the strawmen and power sum quACK to no sidekick connection at all.

Using power sum quACKs increases the packets sent from the proxy to the data sender by 3-9%. These packets either consist mostly of end-to-end ACKs which are sent every packet in `quiche`, or end-to-end ACKs that have been replaced by quACKs in the ACK reduction scenario. We did not evaluate Scenario #1 because it is based on a contrived protocol that lacks many of these features, and the link overheads would not really make sense.

This overhead is representative of the CPU overhead at the client, since quACKs and ACKs take a similar number of cycles to process. In an experiment with Scenario #2 during a period of $\approx$ 90k incoming packets, ACKs took on average 26065 cycles to process while the quACKs took 26369 cycles, 1% more. These cycles come from, i.e., the complex recovery and loss detection algorithms implemented at the end host.

The strawmen have significantly higher link overheads compared to the power sum quACK. The proxy sends up to 10× more packets using Strawman 1a, and also slightly harms the goodput in the congestion control scenario. The reduced goodput is due to the sender mis-identifying received packets as dropped due to dropped quACKs. The proxy achieves higher goodput with Strawman 1b but sends more bytes. Strawman 1c increases the link overheads at both the proxy and the data sender due to larger TCP headers and TCP ACKs. We did not evaluate Strawman 2 due to its impractical decode time.



(a) Low-latency media. CDF of per-packet de-jitter latencies over 10 one-minute trials per protocol.

(b) Path-aware congestion control. Median of 20 trials. Error bars are 1st and 3rd quartiles.

Figure 8: Real-world results. Experiments were run in a moderately well-attended office environment over a Friday afternoon. Trials alternate between the baseline and the sidekick to account for variability in time of day.

6.6 End-to-End Real World Experiments

We discuss the results of our experiments replicating two of our scenarios in the real world, using as context these main differences between emulation and the real-world:

- The RTT is more variable as it depends on interactions in the wireless medium and the shared cellular path.
- Wireless loss can be more variable as nearby 2.4 GHz devices and physical barriers may interfere with the link. Wireless loss also tends to be more clustered in practice.
- The available bandwidth on the shared cellular path is more variable, and depends on the time of day.

Figure 8 shows the results of running the low-latency media and connection-splitting PEP emulation experiments in the real-world. The baseline protocol with a sidekick is able to reduce the 99th percentile de-jitter latency of an audio stream from 2.3 seconds to 204 ms—about a 91% reduction—and improve the goodput of a 50 MB HTTP/3 upload by about 50%. Although the improvements are more conservative compared to emulation in Figure 4a and Figure 4b, each case still benefits the base protocol under all circumstances, compared to end-to-end mechanisms alone.

Part of the difference can be attributed to the network setting. When there is no loss on the near path segment, as can occasionally happen in a real Wi-Fi link, we do not expect to see a difference with a sidekick. When there is more loss on the far path segment, which is variable and depends on the time, we expect the benefit of the sidekick to be less since this equally affects the performance of the base protocol.

The other part of the difference could be made up by future work that better adapts a sidekick connection to real-world variability: The client could improve path segment RTT estimation based on when the proxy receives packets, and use this dynamic estimate in the calculation of r used in β and C. The client could also use this estimate to dynamically adjust the quACK interval. Finally, we could analyze theoretically how PACUBIC responds to traffic patterns in the real world.

7 Limitations

The sidekick approach, and our experiments, are subject to some limitations, which we describe briefly here.

Multipath scenarios. We have only considered sidekick proxies along a single path, and not thought extensively about how quACKs would interact with protocols such as TCPLS [57] that use multiple paths or streams, or even multipath QUIC [18]. To begin thinking about this question, we would have a more complex model of the network: multiple PEPs along a single path, multiple paths each with varying numbers of PEPs, and so on. The proxy can include additional information in the sidekick-reply packet to indicate which path the PEP assistance is on, and the sender can infer from the RTT how far along a path each PEP is relative to others. New sidekick algorithms that come from this model could diagnose troublesome paths, or better allocate network traffic in a multipath connection. Existing algorithms could be applied to individual paths as if they were single-path connections.

Even more diverse network scenarios. The three scenarios we explored all consisted of a lossy Wi-Fi link and a high-latency WAN link. Not all scenarios will be favorable to the sidekick protocol we designed. If the "lossy" section of a network path were on the far path segment from the sender, the sender would not have any more information about the problematic link. To accomodate scenarios like this, sidekick protocols will need more features. For example, the *proxy* would need some way to receive quACKs from the data *receiver*, as well as a mechanism to buffer and retransmit packets [2, 10].

There are likely other scenarios that could benefit from sidekick protocols as described, but we did not evaluate them. For example, if we replaced the lossy Wi-Fi link with a modern wireless link that has a fluctuating physical capacity [9,37,52], the sender may be able to more quickly adapt and make data available for transmission whenever capacity intermittently becomes available.

Practical deployment. The implementation of Robin exists as a research system that has been evaluated in emulation and a limited set of real-world scenarios. Since sidekick protocols require the cooperation of middleboxes and client applications, more work will be needed to standardize the discovery protocol and wire format of sidekick messages described in Section 4, ideally with interest from the IETF. The standards will need to establish several design choices such as how identifiers are computed, how quACKs are transmitted, and the exact mechanisms for security and backwards compatibility. We may also want to standardize sender behavior for specific base protocols, though this could be opaque except to the sender.

The deployment of sidekick protocols can be gradual and backwards-compatible with parties that are either unaware of or do not want to participate in sidekick protocols. To migrate existing client applications, one needs to modify the code to discover a PEP and use information in a quACK to inform the base protocol. To migrate middleboxes, they would need to be modified to listen for sidekick-request markers, then accumulate and send quACKs for participating connections.

Deeper analysis of path-aware congestion control. The correspondence between endpoint-driven PACUBIC and "split CUBIC" is good, and both are better than end-to-end CUBIC in Figure 6), but not exact. The appropriateness of the PACUBIC heuristic, and in general the idea of path-aware congestion control, needs to be further explored. We discuss this more in Appendix A.

8 Conclusion

We presented *sidekick protocols*: an alternate approach to PEPs that leaves the underlying protocol opaque and unmodified on the wire. We described a mathematical technique called a *quACK* that enables middleboxes to refer to packets of the underlying connection without the ability to observe cleartext sequence numbers. We augmented a streaming protocol and a production QUIC implementation (Cloudflare quiche) to make use of information arriving from a proxy on a sidekick connection, including a path-aware congestion-control mechanism called *PACUBIC*. In emulation and a real-world evaluation, the sidekick protocol was able to improve the performance—tail latency, throughput, or energy usage—of these end-to-end base protocols without modifying the wire format or security properties.

Quacknowledgments

We thank our shepherd Akshay Narayan, and Gerry Wan, Deepti Raghavan and Matei Zaharia for their useful feedback. 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, VMware, Dropbox, Amazon, and Meta Platforms.

References

[1] Apple Inc. iCloud Private Relay Overview. https://www.apple.com/icloud/docs/iCloud_Private_Relay_Overview_Dec2021.pdf, Dec. 2021.

[2] H. Balakrishnan, S. Seshan, E. Amir, and R. H. Katz. Improving TCP/IP performance over wireless networks. In *Proceedings of the 1st Annual International Conference on Mobile Computing and Networking*, MobiCom '95, page 2–11, New York, NY, USA, 1995. Association for Computing Machinery.

[3] C. Batut, K. Belabas, D. Bernardi, H. Cohen, and M. Olivier. *User's Guide to PARI-GP*. Université de Bordeaux I, 2000.

[4] BitTorrent Foundation. BitTorrent (BTT) White Paper v0.8.7. https://www.bittorrent.com/btt/btt-docs/BitTorrent_(BTT)_White_Paper_v0.8.7_Feb_2019.pdf, Feb. 2019.

[5] E. Blanton, D. V. Paxson, and M. Allman. TCP Congestion Control. RFC 5681, Sept. 2009.

[6] J. Border. Google QUIC over satellite links. Presentation, IETF PANRG interim, June 2020.

[7] J. Border, B. Shah, C.-J. Su, and R. Torres. Evaluating QUIC's performance against performance enhancing proxy over satellite link. In *2020 IFIP Networking Conference (Networking)*, pages 755–760, 2020.

[8] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker. P4: programming protocol-independent packet processors. *SIGCOMM Comput. Commun. Rev.*, 44(3):87–95, July 2014.

[9] H. Burchardt, N. Serafimovski, D. Tsonev, S. Videv, and H. Haas. VLC: Beyond point-to-point communication. *IEEE Communications Magazine*, 52(7):98–105, 2014.

[10] C. Caini, R. Firrincieli, and D. Lacamera. PEPsal: a performance enhancing proxy designed for TCP satellite connections. In *2006 IEEE 63rd Vehicular Technology Conference*, volume 6, pages 2607–2611, 2006.

[11] Y. Cheng, N. Cardwell, N. Dukkipati, and P. Jha. The RACK-TLP Loss Detection Algorithm for TCP. RFC 8985, Feb. 2021.

[12] D. Clark. The design philosophy of the DARPA internet protocols. In *Symposium Proceedings on Communications Architectures and Protocols*, SIGCOMM '88, page 106–114, New York, NY, USA, 1988. Association for Computing Machinery.

[13] Cloudflare, Inc. Quiche. https://github.com/cloudflare/quiche, Feb. 2024.

[14] B. Cronkite-Ratcliff, A. Bergman, S. Vargaftik, M. Ravi, N. McKeown, I. Abraham, and I. Keslassy. Virtualized congestion control. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, page 230–243, New York, NY, USA, 2016. Association for Computing Machinery.

[15] A. Custura, T. Jones, and G. Fairhurst. Impact of acknowledgements using IETF QUIC on satellite performance. In *2020 10th Advanced Satellite Multimedia Systems Conference and the 16th Signal Processing for Space Communications Workshop (ASMS/SPSC)*, pages 1–8, 2020.

[16] A. Custura, T. Jones, R. Secchi, and G. Fairhurst. Reducing the acknowledgement frequency in IETF QUIC. *International Journal of Satellite Communications and Networking*, 41(4):315–330, 2023.

[17] P. Davern, N. Nashid, C. J. Sreenan, and A. H. Zahran. HTTPEP: a HTTP performance enhancing proxy for satellite systems. *Int. J. Next Gener. Comput.*, 2(3), 2011.

[18] Q. De Coninck and O. Bonaventure. Multipath QUIC: Design and evaluation. In *Proceedings of the 13th International Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '17, page 160–166, New York, NY, USA, 2017. Association for Computing Machinery.

[19] F. R. Dogar and P. Steenkiste. Architecting for edge diversity: supporting rich services over an unbundled transport. In *Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies*, CoNEXT '12, page 13–24, New York, NY, USA, 2012. Association for Computing Machinery.

[20] DPDK: Data Plane Development Kit. https://www.dpdk.org/, Sept. 2023.

[21] K. Edeline and B. Donnet. A bottom-up investigation of the transport-layer ossification. In *2019 Network Traffic Measurement and Analysis Conference (TMA)*, pages 169–176, 2019.

[22] D. Eppstein and M. T. Goodrich. Straggler identification in round-trip data streams via Newton's identities and invertible Bloom filters. *IEEE Trans. on Knowl. and Data Eng.*, 23(2):297–306, Feb. 2011.

[23] V. Farkas, B. Héder, and S. Nováczki. A split connection TCP proxy in LTE networks. In R. Szabó and A. Vidács, editors, *18th European Conference on Information and Communications Technologies (EUNICE)*, volume LNCS-7479, Budapest, Hungary, Aug. 2012. Springer.

[24] B. Ford and J. R. Iyengar. Breaking up the transport logjam. In *Proceedings of the 7th ACM Workshop on Hot Topics in Networks*, HotNets '08, pages 85–90, New York, NY, USA, 2008. Association for Computing Machinery.

[25] P. Goyal, M. Alizadeh, and H. Balakrishnan. Rethinking congestion control for cellular networks. In *Proceedings of the 16th ACM Workshop on Hot Topics in Networks*, HotNets '17, page 29–35, New York, NY, USA, 2017. Association for Computing Machinery.

[26] J. Griner, J. Border, M. Kojo, Z. D. Shelby, and G. Montenegro. Performance Enhancing Proxies Intended to Mitigate Link-Related Degradations. RFC 3135, June 2001.

[27] S. Ha, I. Rhee, and L. Xu. CUBIC: a new TCP-friendly high-speed TCP variant. *SIGOPS Oper. Syst. Rev.*, 42(5):64–74, July 2008.

[28] D. A. Hayes, D. Ros, and O. Alay. On the importance of TCP splitting proxies for future 5G mmWave communications. In *2019 IEEE 44th LCN Symposium on Emerging Topics in Networking (LCN Symposium)*, pages 108–116, 2019.

[29] K. He, E. Rozner, K. Agarwal, Y. J. Gu, W. Felter, J. Carter, and A. Akella. AC/DC TCP: Virtual congestion control enforcement for datacenter networks. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, page 244–257, New York, NY, USA, 2016. Association for Computing Machinery.

[30] M. Honda, Y. Nishida, C. Raiciu, A. Greenhalgh, M. Handley, and H. Tokuda. Is it still possible to extend TCP? In *Proceedings of the 2011 ACM SIGCOMM Conference on Internet Measurement Conference*, IMC '11, page 181–194, New York, NY, USA, 2011. Association for Computing Machinery.

[31] J. Iyengar, I. Swett, and M. Kühlewind. QUIC Acknowledgement Frequency. Internet-Draft draft-ietf-quic-ack-frequency-07, Internet Engineering Task Force, Oct. 2023. Work in Progress.

[32] J. Iyengar and M. Thomson. QUIC: A UDP-Based Multiplexed and Secure Transport. RFC 9000, May 2021.

[33] J. R. Iyengar and B. Ford. Flow splitting with fate sharing in a next generation transport services architecture. *CoRR*, abs/0912.0921, 2009.

[34] A. Kapoor, A. Falk, T. Faber, and Y. Pryadkin. Achieving faster access to satellite link bandwidth. In *Proceedings IEEE INFOCOM 2006. 25TH IEEE International Conference on Computer Communications*, pages 1–6, 2006.

[35] M. G. Karpovsky, L. B. Levitin, and A. Trachtenberg. Data verification and reconciliation with generalized error-control codes. *IEEE Transactions on Information Theory*, 49(7):1788–1793, 2003.

[36] D. Kliazovich, S. Redana, and F. Granelli. Cross-layer error recovery in wireless access networks: The ARQ proxy approach. *Int. J. Commun. Syst.*, 25(4):461–477, Apr. 2012.

[37] S. Koenig, D. Lopez-Diaz, J. Antes, F. Boes, R. Henneberger, A. Leuther, A. Tessmann, R. Schmogrow, D. Hillerkuss, R. Palmer, T. Zwick, C. Koos, W. Freude, O. Ambacher, J. Leuthold, and I. Kallfass. Wireless sub-THz communication system with high data rate. *Nature Photonics*, 7:977–981, Oct. 2013.

[38] M. Kosek, H. Cech, V. Bajpai, and J. Ott. Exploring proxying QUIC and HTTP/3 for satellite communication. In *2022 IFIP Networking Conference (IFIP Networking)*, pages 1–9, 2022.

[39] M. Kosek, T. Shreedhar, and V. Bajpai. Beyond QUIC v1: A first look at recent transport layer IETF standardization efforts. *IEEE Communications Magazine*, 59(4):24–29, 2021.

[40] Z. Krämer, M. Kühlewind, M. Ihlar, and A. Mihály. Cooperative performance enhancement using QUIC tunneling in 5G cellular networks. In *Proceedings of the Applied Networking Research Workshop*, ANRW '21, page 49–51, New York, NY, USA, 2021. Association for Computing Machinery.

[41] Z. Krämer, S. Molnár, M. Pieskä, and A. Mihály. 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)*, pages 1–6, 2020.

[42] N. Kuhn, F. Michel, L. Thomas, E. Dubois, and E. Lochin. QUIC: Opportunities and threats in SATCOM. In *2020 10th Advanced Satellite Multimedia Systems Conference and the 16th Signal Processing for Space Communications Workshop (ASMS/SPSC)*, pages 1–7, 2020.

[43] T. Li, K. Zheng, K. Xu, R. A. Jadhav, T. Xiong, K. Winstein, and K. Tan. Tack: Improving wireless transport performance by taming acknowledgments. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '20, page 15–30, New York, NY, USA, 2020. Association for Computing Machinery.

[44] Y. Li, X. Zhou, M. Boucadair, J. Wang, and F. Qin. 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, July 2020. Work in Progress.

[45] libcurl - the multiprotocol file transfer library. https://curl.se/libcurl/, Sept. 2023.

[46] A. M. Mandalari, A. Lutu, B. Briscoe, M. Bagnulo, and O. Alay. Measuring ECN++: Good news for ++, bad news for ECN over mobile. *IEEE Communications Magazine*, 56(3):180–186, 2018.

[47] A. Martin and N. Khademi. On the suitability of BBR congestion control for QUIC over GEO SATCOM networks. In *Proceedings of the Workshop on Applied Networking Research*, ANRW '22, New York, NY, USA, 2022. Association for Computing Machinery.

[48] S. McCanne and V. Jacobson. The BSD packet filter: a new architecture for user-level packet capture. In *Proceedings of the USENIX Winter 1993 Technical Conference*, USENIX '93, page 2, USA, 1993. USENIX Association.

[49] A. Mihály, S. Nádas, S. Molnár, Z. Krämer, R. Skog, and M. Ihlar. Supporting multi-domain congestion control by a lightweight PEP. In *2017 International Conference on Internet of Things, Embedded Systems and Communications (IINTEC)*, pages 105–110, 2017.

[50] Y. Minsky, A. Trachtenberg, and R. Zippel. Set reconciliation with nearly optimal communication complexity. *IEEE Transactions on Information Theory*, 49(9):2213–2218, 2003.

[51] P. L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44(170):519–521, 1985.

[52] Y. Niu, Y. Li, D. Jin, L. Su, and A. V. Vasilakos. A survey of millimeter wave communications (mmWave) for 5G: opportunities and challenges. *Wireless Networks*, 21:2657–2676, 2015.

[53] G. Papastergiou, G. Fairhurst, D. Ros, A. Brunstrom, K.-J. Grinnemo, P. Hurtig, N. Khademi, M. Tüxen, M. Welzl, D. Damjanovic, and S. Mangiante. Deossifying the internet transport layer: A survey and future perspectives. *IEEE Communications Surveys & Tutorials*, 19(1):619–639, 2017.

[54] C. Perkins, M. Westerlund, and J. Ott. Media Transport and Use of RTP in WebRTC. RFC 8834, Jan. 2021.

[55] M. Polese, M. Mezzavilla, M. Zhang, J. Zhu, S. Rangan, S. Panwar, and M. Zorzi. milliProxy: A TCP proxy architecture for 5G mmWave cellular systems. In *2017 51st Asilomar Conference on Signals, Systems, and Computers*, pages 951–957, 2017.

[56] C. Raiciu, C. Paasch, S. Barre, A. Ford, M. Honda, F. Duchene, O. Bonaventure, and M. Handley. 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, page 29, USA, 2012. USENIX Association.

[57] F. Rochet, E. Assogba, and O. Bonaventure. TCPLS: Closely integrating TCP and TLS. In *Proceedings of the 19th ACM Workshop on Hot Topics in Networks*, HotNets '20, page 45–52, New York, NY, USA, 2020. Association for Computing Machinery.

[58] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-end arguments in system design. *ACM Trans. Comput. Syst.*, 2(4):277–288, Nov. 1984.

[59] J. Sherry, C. Lan, R. A. Popa, and S. Ratnasamy. Blindbox: Deep packet inspection over encrypted traffic. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, page 213–226, New York, NY, USA, 2015. Association for Computing Machinery.

[60] W. R. Stevens. TCP Slow Start, Congestion Avoidance, Fast Retransmit, and Fast Recovery Algorithms. RFC 2001, Jan. 1997.

[61] D. J. D. Touch. Transport Options for UDP. Internet-Draft draft-ietf-tsvwg-udp-options-28, Internet Engineering Task Force, Nov. 2023. Work in Progress.

[62] G. Wan, F. Gong, T. Barbette, and Z. Durumeric. Retina: analyzing 100GbE traffic on commodity hardware. In *Proceedings of the ACM SIGCOMM 2022 Conference*, SIGCOMM '22, page 530–544, New York, NY, USA, 2022. Association for Computing Machinery.

[63] K. Winstein and H. Balakrishnan. Mosh: An interactive remote shell for mobile clients. In *2012 USENIX Annual Technical Conference*, USENIX ATC '12, pages 177–182. USENIX Association, June 2012.

[64] S. Woo and K. Park. Scalable TCP session monitoring with symmetric receive-side scaling. *KAIST, Daejeon, Korea, Tech. Rep*, 144, 2012.

[65] G. Yuan. Quack. https://github.com/ygina/quack, Feb. 2024.

[66] G. Yuan. Sidekick. https://github.com/ygina/sidekick, Feb. 2024.

[67] G. Yuan, D. K. Zhang, M. Sotoudeh, M. Welzl, and K. Winstein. Sidecar: in-network performance enhancements in the age of paranoid transport protocols. In *Proceedings of the 21st ACM Workshop on Hot Topics in Networks*, HotNets '22, page 221–227, New York, NY, USA, 2022. Association for Computing Machinery.

[68] J. Zirngibl, P. Buschmann, P. Sattler, B. Jaeger, J. Aulbach, and G. Carle. It's over 9000: analyzing early QUIC deployments with the standardization on the horizon. In *Proceedings of the 21st ACM Internet Measurement Conference*, IMC '21, page 261–275, New York, NY, USA, 2021. Association for Computing Machinery.

[69] Zoom Video Communications, Inc. Zoom Encryption White Paper. https://explore.zoom.us/docs/doc/Zoom%20Encryption%20Whitepaper.pdf, Aug. 2021.

A Intuitive Analysis of PACUBIC

Here, we dive deeper into the intuition behind the PACUBIC constants (Section 4.3.2), including how they were derived and why the PACUBIC algorithm achieves similar congestion behavior to the CUBIC algorithm in a split connection—we call this behavior "split CUBIC".

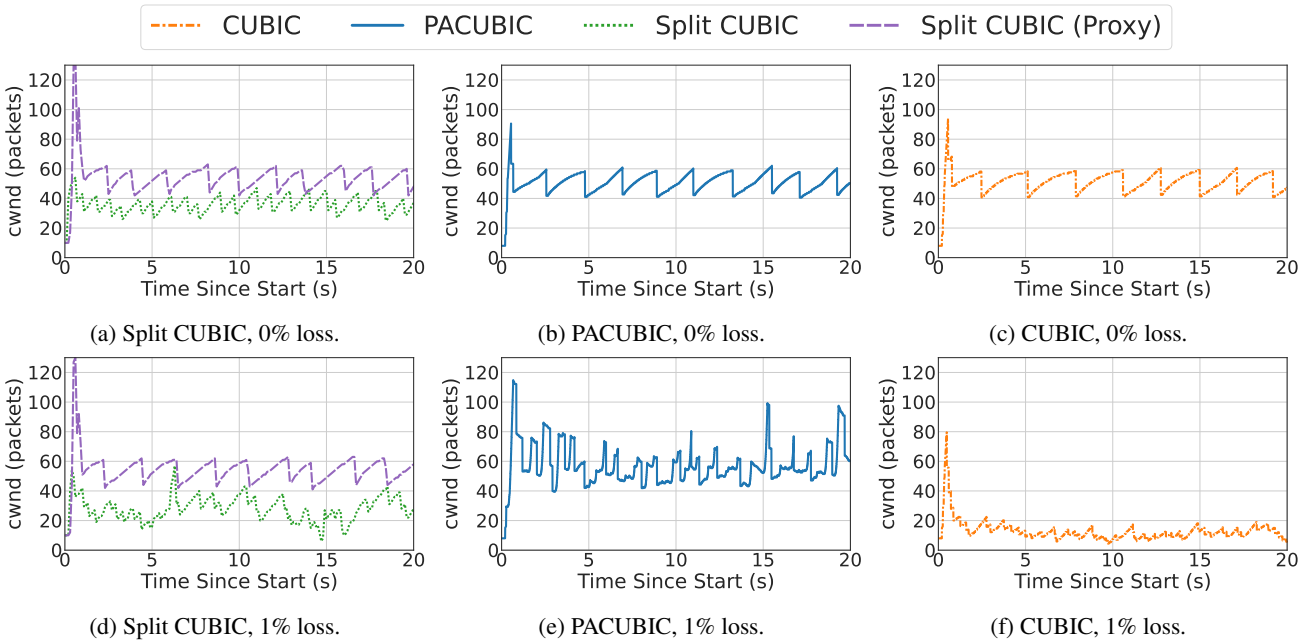
Consider the same network topology as Figure 1 in which a data sender uploads a large file to a data receiver, with help from a sidekick proxy in the middle of the connection. The near path segment connects the sender to the proxy, and the far path segment connects the proxy to the receiver. The near segment is low-delay with varying random loss, and the far segment is high-delay with no random loss. The far segment is the bottleneck link in terms of bandwidth. The actual link parameters are the same as in Scenario #2 of Table 5.

We first discuss how split CUBIC would behave in this setting to conceptually motivate PACUBIC. Consider the congestion windows of each half of the split connection, one taken at the data sender and one at the proxy (Figures 9a and 9d). The far path segment experiences only congestive loss, leading the window at the proxy to fluctuate around the segment's BDP regardless of the loss on the near path segment. The window at the data sender independently determines whether the packets that reach the proxy will be able to fully utilize the window set at the far path segment. The data sender is able to achieve this at low random loss rates, but becomes the bottleneck as loss rates increase (Figure 6).

While split CUBIC has two windows, PACUBIC only has one window representing the in-flight bytes of the end-to-end connection. PACUBIC considers loss detected from both quACKs and end-to-end ACKs. Conceptually, we want an algorithm that would enable PACUBIC's single congestion window to match the sum of CUBIC's two congestion windows, or the total number of in-flight bytes.

With no random loss on the near path segment, PACUBIC (Figure 9b) behaves the same as normal CUBIC (Figure 9c). The congestion window is entirely governed by end-to-end ACKs since the far path segment is the bottleneck link. Note that while the sender may be able to deduce that a loss occurred on the far path segment by combining info from the quACK with the end-to-end ACK, PACUBIC conservatively treats the loss as occurring anywhere on the path.

With some random loss on the near path segment, PACUBIC grows and reduces cwnd based on where the last congestion event occurs (Figure 9e). Note that if the congestion window *cwnd* represents the bytes in-flight in the end-to-end connection, then $r \cdot cwnd$ represents the proportion of bytes in-flight on the near path segment. At a high level, if the data sender discovers loss on the near path segment via the quACK, it holds the $(1-r) \cdot cwnd$ portion of the "far window" constant while applying the CUBIC algorithm to the remaining $r \cdot w_{max}$ of the "near window," representing the bottleneck link.



(a) Split CUBIC, 0% loss. (b) PACUBIC, 0% loss. (c) CUBIC, 0% loss.

(d) Split CUBIC, 1% loss. (e) PACUBIC, 1% loss. (f) CUBIC, 1% loss.

Figure 9: Congestion window of a long-running upload in Scenario #2 (Table 5) with 0% and 1% loss on the near path segmenet. The cwnd is measured at the data sender, except for split CUBIC whose split connection also has a cwnd at the proxy. PACUBIC reacts to every congestion event while keeping the cwnd high. CUBIC performs poorly when there is loss on the near path segment. CUBIC and PACUBIC are implemented in QUIC, while split CUBIC is implemented in TCP using a PEP.

Mathematically, instead of reducing w_{max}, the window size just before the last reduction, by $(1-\beta^*)\cdot w_{max}$, PACUBIC reduces it by only $[1-(1-r(1-\beta^*))]\cdot w_{max} = r(1-\beta^*)\cdot w_{max}$. That is r times the original reduction, a *smaller* amount. We use the RTT ratio r (near path segment to end-to-end) as a proxy for the ratio of the number of in-flight bytes.

Similarly, instead of using a cubic growth function with scaling factor C^* and inflection point $K = K^* = \sqrt[3]{w_{max}(1-\beta^*)/C^*}$, we use a larger scaling factor $C = C^*/r^3$ and thus a shorter inflection point

$$K = \sqrt[3]{\frac{w_{max}(1-\beta)}{C}} = \sqrt[3]{\frac{r\cdot w_{max}(1-\beta^*)}{C^*/r^3}} = r^{4/3}\cdot K^*.$$

The shorter inflection point leads the congestion window to *grow more quickly* since the sender also reacts to feedback about loss more quickly over the low-delay link.

At times, there can be loss detected both in quACKs and in end-to-end ACKs. The end-to-end ACKs have a greater effect since they reduce the congestion window by a larger proportion, until the remaining path segment with loss is the bottleneck link. In this scenario with loss, the bottleneck link at equilibrium is the near path segment. At this point, the quACK primarily determines the congestion window updates. If the far path segment were to become the bottleneck again, the data sender would detect a congestion event via the end-to-end ACK.

PACUBIC has several limitations. Although it beats end-to-end CUBIC, it still performs worse than split CUBIC, especially at high loss rates (Figure 6). Also, it doesn't consider loss on the far path segment any differently than original CUBIC, unlike split CUBIC which treats the two split connections independently. PACUBIC emulates the congestion control behavior and fairness of split CUBIC fairly well as a heuristic, but would benefit from an analysis in a wider variety of network scenarios. It would also benefit from a side-by-side fairness comparison against other congestion control algorithms that perform well in the same scenarios. We'd like the primary takeaway of PACUBIC to be that knowing where loss occurs can cleverly inform congestion control.