In-Network Retransmissions for Encrypted Transport Protocols

GINA YUAN*, Stanford University, USA
THEA ROSSMAN*, Stanford University, USA
MICHAEL WELZL, University of Oslo, Norway
KEITH WINSTEIN, Stanford University, USA

This paper describes and evaluates new techniques for network-originated retransmissions for end-to-end transport connections, yielding performance benefits for encrypted transport protocols in lossy settings. We use set-reconciliation techniques based on the Rateless IBLT to let receivers efficiently acknowledge encrypted packets to a middlebox, without modifying the sender or the underlying wire format. The scheme integrates awareness of in-network retransmissions within transport receivers, which delay some selective-acknowledgment ranges to reduce spurious end-to-end retransmissions. With these tools, transport receivers can receive protocol-agnostic, network-originated retransmissions for encrypted transport connections.

CCS Concepts: • **Networks** → **Network protocol design**; **Middle boxes / network appliances**; *Transport protocols*; *In-network processing*; Network reliability.

Additional Key Words and Phrases: Performance-Enhancing Proxies, Encrypted Transport Protocols

ACM Reference Format:
Gina Yuan, Thea Rossman, Michael Welzl, and Keith Winstein. 2026. In-Network Retransmissions for Encrypted Transport Protocols. *Proc. ACM Netw.* 4, CoNEXT2, Article 16 (June 2026), 22 pages. https://doi.org/10.1145/3808664

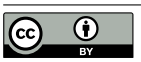
1 Introduction

The relationship between Internet transport protocols and lossy networks has long been fraught. Since the rise of wireless Ethernet (Wi-Fi), researchers and operators have observed that TCP's end-to-end reliability mechanisms perform poorly over paths with substantial non-congestive, IP-layer packet loss [5, 12, 15, 31, 43]. Most congestion-control schemes, including the popular BBRv3 [11] and CUBIC [25], interpret packet loss as a sign of congestion and slow down in response—a mistake when the loss is caused by, for instance, physical-layer interference. TCP and QUIC connections can become essentially unusable when IP-layer loss reaches 5% over high-latency paths [4, 16, 42, 58].

Because of this, many wireless links use link-layer acknowledgments and retransmissions to mask loss from the IP layer [1, 29, 38, 46]. However, without knowledge of transport or application preferences, there is a limit to the "effort" that a reliable link can employ. A link that reliably delivers every packet *in order* will create unacceptable delays for latency-sensitive flows, because delivered packets must wait for earlier lost packets to be retransmitted [20, 56]. A reliable link that delivers every packet when it arrives (out of order) can trigger spurious transport-layer retransmissions, especially when the link's latency is an appreciable fraction of the end-to-end RTT [13, 30].

*Both authors contributed equally to this work.

Authors' Contact Information: Gina Yuan, Stanford University, USA; Thea Rossman, Stanford University, USA; Michael Welzl, University of Oslo, Norway; Keith Winstein, Stanford University, USA.



ACM 2834-5509/2026/6-ART16
https://doi.org/10.1145/3808664

As a result, reliability mechanisms at the link layer are necessarily tuned to be acceptable as a one-size-fits-all, and some non-congestive loss remains visible to the IP layer. To mitigate the effects of this on TCP, network operators have long deployed performance-enhancing proxies (PEPs) to accelerate end-to-end connections. These include TCP connection-splitting proxies, which transparently create two concatenated connections from a single end-to-end connection [10, 21, 24, 32]; and middleboxes that passively observe and retransmit packets [5, 14, 43, 50]. Such middleboxes have been widely criticized. While they can improve application-layer performance [26, 32, 48, 58], they can also hinder protocol evolution, add complexity, and introduce performance bottlenecks as links become faster [17, 27, 47]. Additionally, these techniques simply do not work with encrypted transport protocols such as QUIC [37] without terminating encryption in the network. In some situations, QUIC connections can experience significantly lower transport-layer performance than TCP connections that, over the same path, are assisted by a PEP [7, 8, 33, 36, 41, 54, 58].

In 2024, Yuan et al. proposed *sidekick protocols* [59]: an approach to in-network assistance where an end-host and performance-enhancing proxy exchange information about an underlying, or "base", connection over an adjacent channel. The sidekick protocol is agnostic to the base connection, which remains opaque and unmodified. Yuan et al. present one such sidekick protocol, "Robin," in which an in-network proxy sends acknowledgments to a data sender, helping it quickly retransmit lost packets and perform appropriate congestion control.

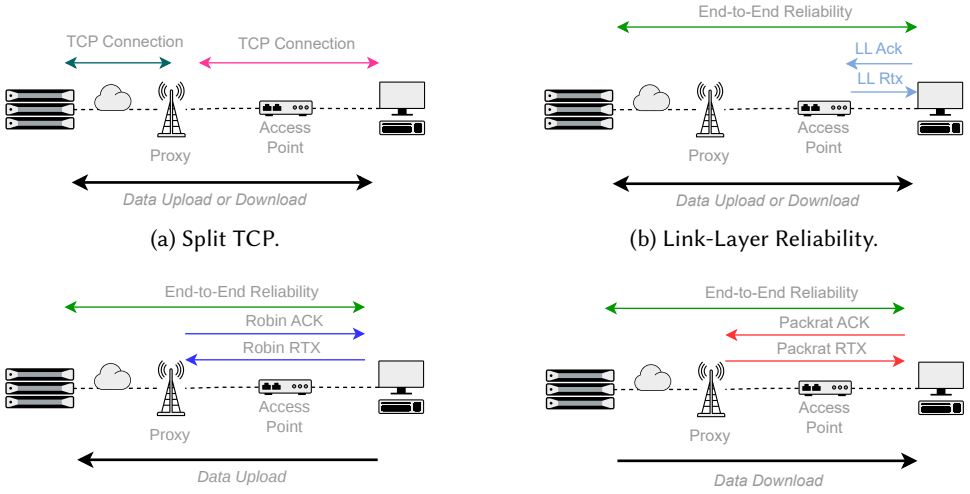
However, Robin faces a key limitation: the proxy only sends in-network *acknowledgments* for a data *sender*, not in-network *retransmissions* for a data *receiver*. As a result, this technique is useful only when the lossy segment is near the data sender. As the last-mile segment (closer to a user) is typically the lossy one, Robin assists *upload* traffic patterns. Can a sidekick approach help *downloads*, where the lossy segment is near the data receiver?

In this paper, we describe and evaluate a new sidekick protocol that provides in-network *retransmissions* for encrypted transport connections. We call this protocol and associated proxy Packrat[1]. In emulation measurements, Packrat-aided connections approximate the performance of connections aided by connection-splitting PEPs. Packrat is deployed in two places: a proxy in the network, and a user-space receiver application, e.g., linked into a receiving Web browser or video-conferencing client. Packrat does not require modifications to the data sender (e.g. a webserver) or to the transport protocol's wire format or to the link-layer protocol or implementation. This paper makes two principal contributions:

- We present a new construction of an acknowledgment for encrypted packets based on the Rateless Invertible Bloom Lookup Table (rIBLT) [57]. Unlike in Robin, the sidekick protocol proposed in [59], the encrypted acknowledgments in Packrat are decoded by an in-network proxy that may be handling many such connections. The rIBLT-encrypted ACK is much faster to decode than the construction used in Robin, allowing Packrat to scale to proxies that handle 10 Gbit/s per core.
- Unlike in Robin, a Packrat connection has two sources of retransmissions: the proxy and the sending endpoint, the latter of which is unmodified and unaware of Packrat. We describe a new interface between the transport-protocol receiver (e.g. a QUIC receiver) and the adjacent proxy-to-receiver connection (the Packrat connection) that prevents spurious retransmissions and congestion response. The transport-protocol receiver edits its end-to-end acknowledgments to withhold ACK ranges that would spuriously trigger a sender-side loss response before the Packrat proxy has had an opportunity to retransmit the same packet.

We integrated the Packrat protocol with three applications and show that it can enable a variety of performance enhancements given a lossy path segment near the data receiver. In emulation

[1]For "packet rateless retransmission"—and because the proxy keeps a cache of packets-in-flight for possible retransmission.



(a) Split TCP.

(b) Link-Layer Reliability.

(c) Robin, the sidekick protocol proposed in [59].

(d) Packrat, the sidekick protocol proposed in this paper.

Fig. 1. In-network retransmission mechanisms, including Robin (in-network acknowledgments from a proxy to a data sender) (Figure 1c), and Packrat (in-network retransmission from a proxy to a data receiver) (Figure 1d). Each proxy and client communicate over a lossy, last-mile segment (one or more links).

experiments, we found that Packrat improved throughput for a large file download using QUIC and HTTP/3; reduced tail latency of a media stream with forward error-correction (FEC); and reduced end-to-end retransmissions of a reliable multicast real-time transport stream. We open-source our implementation of the Packrat proxy, rIBLT-based acknowledgment library, and client integrations.[2]

2 Related Work

We provide a brief overview of in-network loss recovery mechanisms to motivate the need for protocol-agnostic in-network retransmissions.

Performance-Enhancing Proxies Networks accelerate TCP connections with connection-splitting PEPs (Figure 1a) at the boundary between the wired Internet and a lossy, wide-area subpath [7, 10, 17, 21, 24, 32, 47]. These PEPs transparently interpose on TCP connections that cross them, creating two concatenated connections (not restricted to an individual link). While controversial, splitting a TCP connection can improve application-layer performance over end-to-end paths that include a lossy segment [9, 21, 26, 32, 48, 58]. Intuitively, connection-splitting reduces the latency of each congestion control feedback loop, allowing both the sender and proxy to more quickly recover from loss. However, connection-splitting PEPs rely on plaintext TCP sequence numbers and limit the deployment of TCP options and new behaviors [7, 27, 47, 51]. They share fate with endpoints (if a PEP goes down, the connection must reset) and can create performance bottlenecks. Packrat is spiritually similar to Snoop [5], which observes, buffers, and retransmits TCP packets to mask loss—a less intrusive mechanism than connection splitting. However, Snoop relies on plaintext TCP sequence numbers and is inapplicable to encrypted transports.

Lower-level acknowledgment and retransmission. Most wireless link technologies use reliability mechanisms that acknowledge and retransmit packets, up to a retry threshold [20, 35, 46] (Figure 1b). Link-layer receivers typically put packets back in order before releasing them to IP, up to some

[2]https://github.com/StanfordSNR/sidekick-downloads/

timeout or buffer limit [1, 20, 29, 38, 46, 56]. These mechanisms are necessary to make TCP and QUIC practical over wireless links, but the retry and reordering thresholds are necessarily limited by the need to accommodate both reliable transport connections and latency-sensitive transport connections. PEPs, Robin [59], and Packrat are meant to deal with the remaining non-congestive *IP-layer* loss.

Some VPNs and tunnels can reliably transmit encapsulated IP datagrams, similar to a reliable link layer; MASQUE proxies tunnel IP traffic over QUIC [3, 34, 52, 53]. The relationship between end-to-end and proxy-to-endpoint reliability and congestion control in MASQUE deployments is under active development.

Sidekick Protocols and Robin. Yuan et al. introduced sidekick protocols [59]: in-network assistance mechanisms that execute adjacent and agnostic to the base connection. They implemented one such sidekick protocol, "Robin" (Figure 1c). With Robin, an in-network proxy sends acknowledgments to a data sender, helping it quickly retransmit lost packets and perform appropriate congestion control. A key contribution of this work was a mathematical construct—the "quACK"—which lets proxies acknowledge encrypted packets to a data sender without sequential, cleartext sequence numbers [59]. Yuan et al. modeled the problem of providing a concise acknowledgment of encrypted packet identifiers as a set-reconciliation problem [19, 44]. The Robin system solved a system of power sum polynomial equations to recover a set difference.

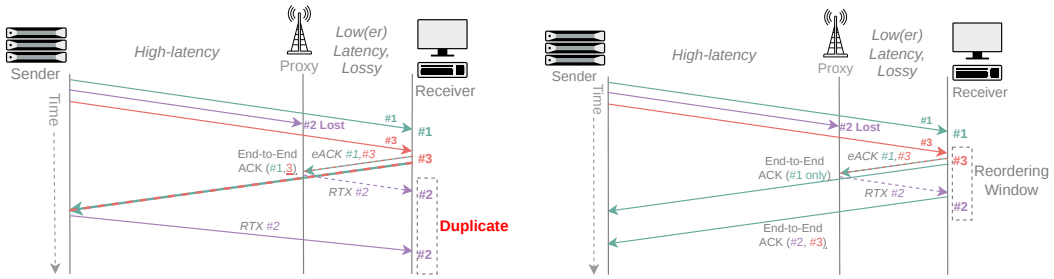
As noted in Section 1, Robin supports a typical *upload* traffic pattern: the proxy sends quACKs to signal loss to a data sender, which retransmits along the end-to-end connection. A naive approach to protocol-agnostic, network-originated retransmission would simply "flip" these roles. However, adapting Yuan et al.'s work to downloads presents three new challenges. First, the in-network proxy must cache and retransmit packets, not just acknowledge them. Second, any in-network retransmission mechanism must address potential in-network reordering and associated spurious retransmissions (Section 4). Finally, the system must optimize decoding of encrypted ACKs. In Robin, the proxy only needed to *encode* quACKs. To support in-network retransmission, it would also have to decode them. For the power sum quACK introduced in Robin, decoding is significantly more expensive than encoding (Section 5). While Packrat is a "sidekick protocol for downloads", providing in-network retransmissions is a different problem that introduces new challenges not addressed by Robin.

3 Packrat Overview

We begin with an overview of the key components of the Packrat protocol, client, and proxy. The Packrat protocol is spoken between a client and an on-path proxy, where the client is a data *receiver*. We imagine the proxy to exist near the data receiver, peered across a network segment with high non-congestive loss. Use of the proxy is entirely optional for correctness, as the base connection can fall back on end-to-end retransmissions.

Steady-State. After an initial handshake to negotiate parameters, the Packrat client and proxy exchange data on a (separate) Packrat connection to negotiate in-network retransmissions. The client and proxy each track the set of all locally observed packets in a compact data structure (Section 5), which we refer to as an *encrypted acknowledgment* (eACK).[3] In addition to managing the end-to-end connection, the client periodically transmits this eACK to the proxy, indicating the packets it has received. The proxy passively observes and records packets from the base connection in its eACK data structure and in a local retransmission cache. The proxy uses eACKs received from

[3]We avoid reusing the "quACK" terminology introduced by Yuan et al. [59] to reflect the eACK's different construction and use in Packrat (Section 5).



(a) Without selectively withheld acknowledgment gaps: the proxy retransmits packet #2, but it doesn't arrive before the receiver sends an end-to-end ACK that causes the sender to retransmit the same packet.

(b) With selectively withheld acknowledgment gaps: the receiver modifies the transport SACK to exclude #3, giving the proxy an opportunity to retransmit #2 before the sender learns of the loss.

Fig. 2. Any in-network retransmission scheme must recover lost packets before the end-to-end transport connection triggers a retransmission and associated congestion response. Packrat receivers selectively withhold SACK ranges in a sliding reordering window proportional to an estimated client-to-proxy RTT, waiting to signal loss until the proxy has had a chance to retransmit.

the client to identify missing packets and sends appropriate retransmissions. It evicts packets from its cache when the client acknowledges them or the cache size exceeds a configured threshold.

Resetting. The proxy or client can reset the Packrat connection on an error, beginning a new set reconciliation epoch. We note that, unlike connection-splitters and VPNs, which fate-share with endpoints, base connections that use the Packrat protocol can always fall back on end-to-end retransmissions. In addition to providing a high error tolerance, the Packrat style of in-network assistance is well-suited to mobility, as a connection can change network paths.

4 Preventing Spurious Retransmissions

To avoid a spurious end-to-end retransmission and associated congestion response, any in-network retransmission scheme must either (1) intercept and reorder packets at the receiver (Section 2) or (2) recover dropped packets before the transport sender detects them as "lost." In this section, we describe the reordering problem, existing solutions to it, and the Packrat approach to bringing the transport receiver "into the loop" of in-network retransmission. Specifically, a Packrat client selectively withholds acknowledgment ranges in a sliding, receiver-side reordering window when it detects loss, waiting to signal loss to a sender until the proxy has had a chance to retransmit. We demonstrate in Section 7.3.1 that this mechanism successfully masks end-to-end loss.

4.1 In-Network Reordering

The problem of in-network reordering is well-known in link-layer and transport protocol design [13, 56]. Consider a (naive) Packrat proxy in the simple scenario in Figure 2a. Packet #2 is lost in-flight, but packet #3 is not. The proxy later recovers packet #2, but it arrives after the receiver has indicated it lost. Whether the end-to-end connection interprets this gap in consecutive received sequence numbers as loss depends on the retransmission latency (how quickly was packet #2 recovered?) and the transport protocol's loss detection mechanism (how much "grace" do the receiver and sender allow before inferring loss?).

Most transport senders allow for some in-network reordering [13, 30]. A QUIC sender using the RFC's parameters will infer loss after three out-of-order ACKs or a temporal reordering threshold of 9/8× the estimated end-to-end RTT [30]. For such a sender, an unordered in-network retransmission will fail to mask loss if the lossy segment accounts for more than 1/8× the end-to-end RTT or

reorders >3 packets. RACK-TLP [13, 55] addresses reordering by adapting its time threshold in response to observed network behavior.

4.2 Packrat: Receive-Side Delayed Acknowledgment

The Packrat approach exposes in-network retransmissions to the transport receiver and involves it in the management of in-network reordering. Packrat introduces a receiver-side reordering window, waiting to signal a packet as "lost" until the proxy has had a chance to retransmit. In other words, the end-to-end receiver delays the acknowledgments it sends to the end-to-end sender based on knowledge of the auxiliary connection.

The exact implementation of this depends on the acknowledgment scheme. In a SACK approach typical of QUIC [30], Packrat modifies each SACK to exclude acknowledgment ranges received within the most recent reordering window *if* acknowledging those packets would lead to a sender-side retransmission. That is, Packrat waits to reveal a sequence "gap" to the sender until it has been present for at least one reordering window. Figure 2b depicts the same scenario as Figure 2a with Packrat's acknowledgment delay. Here, the receiver waits to acknowledge packet #3 — signaling the loss of #2 — until after one estimated receiver-to-proxy RTT. If the proxy is able to recover the lost packet, the receiver never signals this loss to the sender, preventing a loss response and spurious retransmission.

5 Scalable In-Network Retransmissions

In the steady-state, the Packrat client and proxy each maintain a local copy of a data structure, the eACK, representing the set of observed packets. The Packrat client sends this eACK to communicate its local packet set at regular intervals, and the proxy identifies and retransmits missing packets. Implementing the eACK requires identifying packets in the base connection without access to cleartext sequence numbers or in-network ordering guarantees.

As described in Section 2, Yuan et al. [59, 60] introduced the *quACK*: a mathematical tool that concisely represents a selective acknowledgment of opaque, randomly-identified, unordered packets. While the Packrat proxy encounters the same encrypted acknowledgment task as Robin, the sidekick protocol introduced in [59], in-network *retransmission*—vs. in-network *acknowledgment*—presents a different set of scalability constraints. Unlike the Robin proxy, the Packrat proxy must encode every packet in the base connection *and* decode every acknowledgment. An on-path proxy handles tens to hundreds of thousands of concurrent connections at once, and any per-packet overhead incurred by the Packrat protocol at the proxy must be extremely small.

In this section, we describe a new construction for eACKs based on the rateless invertible Bloom filter (rIBLT) [57]. We benchmark the rIBLT eACK against a power sum eACK based on the quACK, finding that the former is often faster to decode in practice (1–476× in a sample of network conditions), particularly for high-throughput connections. We find that the rIBLT eACK is better suited to the scalability requirements of in-network retransmission, particularly under high loss or latency, than a power sum approach.

5.1 rIBLT Overview

We provide a brief overview of the key properties of the rIBLT data structure. We direct the user to the original papers for a full description of the IBLT [23] and rIBLT [57].

The Invertible Bloom Lookup Table (IBLT) adapts the classical Bloom filter [6] for compactly encoding and testing set membership [23]. The IBLT data structure maps set items to a small number of symbols in an array. Each symbol encodes an *XOR* of element identifiers and a count of the number of elements mapped to that symbol. The IBLT supports insertion, deletion, and decoding (to recover a set difference). That is, to perform set reconciliation, Alice sends an IBLT

built (encoded) from set A, and Bob recovers the set difference (decodes) by "deleting" the elements in his set B from it. Importantly, encoding, decoding, and communication cost scale with the size of the set *difference*. In the IBLT, decoding is probabilistic: decoding can fail if Alice transmits an insufficient number of symbols to Bob.

The *rateless* IBLT (rIBLT) [57] introduces a novel pseudorandom mapping algorithm, which reduces the memory, encoding, decoding, and communication cost of the IBLT (we explore rIBLT scalability in Section 5.3). The rIBLT is additionally "rateless" in that a prefix of an rIBLT fully represents the rIBLT: each party incrementally encodes its set into a stream of symbols until the other can decode them. The authors show that, for a true number of missing elements m, decoding requires $1.35m$ coded symbols on average. We leverage the efficient mapping, low communication cost, and rateless properties of the rIBLT to construct Packrat encrypted acknowledgments.

5.2 rIBLT Acknowledgment

The eACK data structure implements rIBLT insertion (encoding), deletion, and decoding (to recover a set difference). The client and proxy each incrementally update a local eACK (rIBLT), representing the packets each has observed. At some eACK granularity (Section 6), the client serializes and transmits an updated representation of its rIBLT to the proxy, which calculates the set difference and triggers appropriate retransmissions. In rIBLT terms, the Packrat eACK granularity is its set reconciliation interval. Note that we use "eACK" to refer to both the local data structures that each peer maintains and the serialized representation that the client transmits over the Packrat connection.

5.2.1 Parameterization. Prior work applies the rIBLT to large distributed systems, and parties perform set reconciliation over multiple RTTs until successful decoding. The Packrat setting requires set reconciliation at millisecond timescales, making multi-RTT negotiation impractical. Thus, Packrat peers must preselect parameters for their local rIBLTs.

If the proxy fails to decode an eACK, the Packrat connection temporarily falls back on end-to-end retransmissions. We aim to parameterize the eACK to minimize decoding failure probability while balancing computation and communication cost. Decoding can fail due to identifier collisions, integer (count) overflow, or insufficient symbols, all of which depend on the true number of missing packets per transmitted eACK. Correctly parameterizing local eACK data structures requires estimating an upper bound on missing packets per eACK.

Optimal parameters depend on the throughput of the base connection, eACK granularity, loss rate, error tolerance, and resource constraints. The client and proxy jointly select parameters during an initial handshake (Section 6). Packrat peers configure *local* eACKs for a worst-case scenario, and the client leverages the rateless property of the rIBLT to dynamically adjust each *transmitted* eACK size based on its known number of missing packets (Section 5.2.2).

Symbol Size. If two packets encoded in the same transmitted eACK share an identifier and one is lost, decoding can fail. By default, we conservatively select a four-byte identifier drawn from a fixed offset in the (encrypted, and thus high-entropy) payload. Reaching even a 0.001% collision probability would require 100 lost packets in a single millisecond-scale eACK interval. We choose an eight-bit integer for the `count` field, which represents the number of symbols encoded per cell. Decoding will only fail if more than 255 *missing* packets are encoded in *all of* the same symbols.

Number of Local Symbols. We estimate the number of symbols required to successfully decode an eACK in Figure 4. An rIBLT with more symbols is more likely to be decoded successfully, but it incurs higher encoding, decoding, and communication cost. Because encoding and decoding are relatively cheap (Section 5.3), each Packrat peer can configure its *local* eACK data structure for a

worst case. For our evaluations, we select 160 local symbols for a high-throughput download and 40 for a lower-rate media stream.

5.2.2 Rateless eACKs. While the client and proxy may locally encode a "large" rIBLT, the client need not *transmit* all symbols in every eACK. Because the rIBLT is rateless, a prefix of an rIBLT fully represents the rIBLT. In theory, the client can correctly communicate its t-symbol local rIBLT by transmitting any $t' < t$ symbols to the proxy. While this smaller t' comes at the cost of decoding success probability, it may be sufficient if the true number of currently-missing packets—known to the client—is small.

We thus leverage the rateless property of the rIBLT as follows. From its base connection, the client estimates the number of retransmissions that it expects from the proxy. Based on this number, it selects the appropriate eACK size to transmit based on a desired success probability. We show in Section 7.3.2 that this reduces link overhead.

5.2.3 Selective eACKing. If the client does not expect it needs a retransmission, it need not send an eACK. In NACK schemes, the client can choose to eACK only when it would otherwise send a NACK. Because the proxy optimistically evicts packets, the client can omit regular eACKs without an exploding cache size at the proxy. We integrate selective eACKs in the NACK layer of our media streaming benchmark (Section 6.2) and show in Section 7.3.2 that it reduces link overhead.

5.3 rIBLT vs. Power Sum Microbenchmarks

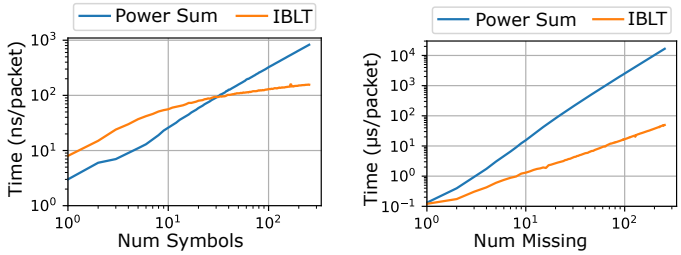
As noted in Section 2, Robin [59] leveraged a set reconciliation approach based on power sums to address the problem of in-network acknowledgments: the in-network proxy encodes symbols into an acknowledgment, representing packets it has received, while the client decodes this acknowledgment to initiate retransmissions. To implement in-network *retransmissions*, can we simply leverage the power sum construction and swap these roles?

We adapt the power sum quACK from [59] to our eACK and directly benchmark it against our rIBLT implementation. In this section, we compare the overheads of the two approaches in practice. We run these microbenchmarks on a single core of an AWS m4.xlarge instance. We observe that rIBLT decoding is faster in practice, especially for connections with relatively high or bursty throughputs. In the power sum approach, decoding is more expensive than encoding and scales linearly with the *total* number of packets that are in the proxy's cache when it receives a new eACK. rIBLT decoding, in contrast, scales logarithmically with the number of *missing* elements. rIBLT and power sum encoding costs are comparable.

Encoding. Both eACK implementations select an initial symbol count, which we call t. While the power sum eACK encodes each item in all t symbols, the rIBLT's encoding algorithm is $O(\log(t))$. However, in practice, each update in the rIBLT uses an expensive square root instruction, adding constant factor overheads that impact smaller numbers of symbols. This difference is on the order of tens of nanoseconds (Figure 3a).

Decoding. Decoding in the rIBLT is faster in practice for any number of symbols (Figure 3b). Note that the power sum eACK actually uses a decoding method that is linear in *all* packets forwarded within an eACK interval, not just the *missing* packets, due to the complexity of symmetric polynomial factorization. We measure the decoding cost per missing packet for a fixed 2% loss rate in Figure 3b, and we translate this to a set of concrete network conditions in Table 2.

Non-determinism and communication cost. In the power sum eACK, the number of symbols required to successfully decode a single eACK is equal to the number of missing packets. In the rIBLT, the number of symbols t required to decode an rIBLT eACK is a probabilistic multiplier x of the actual number of errors m ($x = 1.35$ on average in [57]). The client and proxy can select



(a) Encode time. (b) Decode time.

Fig. 3. The measured time to encode (3a) and decode a single packet (3a) in the rIBLT and power sum eACK for an initial number of symbols t and true number of missing packets m. While the rIBLT is more scalable (Table 1), it incurs constant, nanosecond-scale overheads that impact small set encoding.

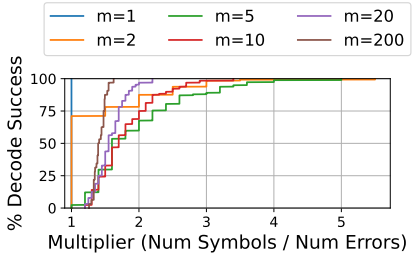


Fig. 4. The number of symbols t required to decode an rIBLT eACK is a constant multiplier x of the actual number of missing packets m. That is, num_symbols = num_errors × multiplier. CDF of multiplier x required to decode an rIBLT-based eACK with probability p (y-axis). 100,000 trials.

	PSum	rIBLT
Encode	$O(t)$	$O(\log(t))$
Decode	$O(Nt)$	$O(m\log(t))$

Table 1. The computational complexity of encoding and decoding a single packet for the power sum [59] and rIBLT eACK. m is the actual number of missing packets, t is the initially-configured number of symbols (based on an upper bound on m), and N is the number of elements in the local set.

Connection Throughput	Loss Burst	eACK RTT	rIBLT Gain
20Mbps	<1%	10ms	1.77–2.15×
20Mbps	15%	10ms	1.00×
50Mbps	50%	10ms	2.03×
20Mbps	10%	100ms	8.09×
100Mbps	<1%	100ms	44.3–476.3×

Table 2. The overhead of decoding a power sum vs. rIBLT eACK from Figure 3b translated to concrete connection and network characteristics. Compared to the power sum, the rIBLT eACK is faster to decode when there are more set items (not-yet-acknowledged packets) during set reconciliation.

high values for their *local* eACKs, yielding a higher worst-case tolerance, but the client *transmits* a prefix of this eACK parameterized to the number of currently-missing packets (the rateless eACK described in Section 5.2.2). Figure 4 shows the CDF of the minimum number of symbols to decode various m as this constant multiplier increases. For example, if 20 packets are missing, the client should send $\approx 1.8 \times 20 = 36$ symbols (rIBLT cells) for decoding to succeed with 80% probability. The Packrat proxy can reset its connection if it cannot decode an eACK.

6 Implementation

We now describe our prototype of the Packrat protocol, which includes the `eack` library, a `packrat` client library used for three client integrations, and a Packrat proxy. We also describe the implementations of the applications we use to evaluate Packrat.

6.1 Packrat Protocol

Connection Establishment. Our Packrat client discovers an on-path proxy by sending a magic `Discovery` packet along the established base connection. A proxy responds with its own `Init` packet, advertising its supported configurations. The client accepts assistance from a proxy by replying with an `Init`, including requested configurations. The proxy accepts or rejects these configurations with an `InitACK`, completing the handshake, and opens a UDP socket for the Packrat connection that will assist this base connection. During establishment, peers agree on the eACK encoding (our implementation uses a fixed byte offset) and the number of symbols in each local rIBLT encoding (Section 5).

After sending an `InitACK`, the proxy begins to cache packets and retransmit. On receiving an `InitACK`, the client begins sending eACKs. In our prototype, the proxy synchronizes starting sets by bridging (rather than sniffing) packets in the end-to-end connection and ordering its `InitACK` with initial data packets.

Client. The Packrat client library exposes APIs to (1) record a received packet; (2) send an eACK; and (3) configure, tear down, and reset the Packrat connection. We invoke this library within each transport (e.g., QUIC) receiver described in Section 6.2. Embedding the Packrat logic in the transport receiver allows us to implement the client "hints" described in Section 5.2.2 and the in-network reordering delay described in Section 4.

Proxy. We implement the Packrat proxy as a network bridge that uses raw sockets to read and write packets between two interfaces. For each connection, the proxy must maintain (1) a cache of forwarded but unacknowledged packets and (2) its locally-stored eACK (i.e., rIBLT representation). We evaluate the memory footprint of the proxy in Section 7.3.3.

The Packrat protocol does not enforce a specific loss detection scheme. In our implementation, the proxy triggers a retransmission upon receiving an unacknowledged gap in the ordered sequence of cached packets. As with end-to-end loss detection schemes, an alternate implementation could incorporate packet time thresholds.

Cache Management. The proxy caches packets from the base connection in the order forwarded to the client up to the preconfigured cache size. It evicts packets from its cache if an `eACK` indicates they have been received and no longer need to be retransmitted. If a packet of the base connection arrives but the cache is full, the proxy *optimistically* evicts the oldest packets. When evicting a packet—whether due to acknowledgment or cache overflow—the proxy encodes it in the locally-stored eACK. Enforcing a cache capacity through optimistic eviction allows the client to eACK less frequently when retransmissions are uncommon (selective eACKs, Section 5.2.3). Evicting an unacknowledged packet from the cache may cause the connection to fall back on an end-to-end retransmission. (We evaluate cache utilization in Section B.)

Resetting the Packrat Connection. If the proxy is unable to decode an eACK, it sends a `Reset` to the client. Decoding may fail if (1) there are an insufficient number of symbols to decode missing packets (recall from Section 5 that the rIBLT eACK is probabilistic); (2) the proxy optimistically evicted a necessary retransmission (this causes the client and proxy data structures to become out-of-sync); or (3) a packet corruption or collision caused a mismatch in encoded identifiers. The

`Reset` indicates the start of a new epoch in which the client and proxy encode identifiers into a new eACK data structure. That is, each eACK is the cumulative representation of all packets received since either the start of the connection or the last `Reset`.

Multicast Proxy. We implement an extension to the Packrat proxy to assist IP multicast, or, more generally, multiple clients that share a single base data stream. The proxy maintains a single fixed-size cache for the multicast four-tuple. For each client, the proxy maintains an eACK and a virtual cache. The virtual cache contains (1) a global index in the base cache of the client's first unacknowledged packet and (2) pairs of global indexes that indicate which packet to insert and where to insert it because it was retransmitted to the client. This allows the proxy to maintain state proportional to the number of outstanding retransmissions per client.

6.2 Client Applications

We evaluate the Packrat protocol in three applications with different performance metrics to explore the versatility of in-network retransmissions: a high-throughput HTTP/3 file download, a low-latency media stream with forward error correction, and a reliable multicast stream whose server has limited capacity to handle end-to-end unicast retransmissions.

HTTP/3 file download. We measure the goodput of a large file download over HTTP/3, using the default client and server in the Picoquic QUIC implementation [28]. Picoquic is an open-source QUIC implementation with simplicity and RFC compliance as first-order goals. The client requests 25MB of random data, and we measure goodput by dividing the client-side connection time by the requested data size, excluding headers. We run experiments using Picoquic's implementations of both CUBIC [25] and BBRv3 [11], though we caution that the behavior of congestion control schemes varies by implementation [45, 58].

Integrating Packrat with the Picoquic client required 254 lines of code. Packrat requires a socket loop that performs discovery and sends regular eACKs to the proxy, as well as methods that intercept, in order to selectively delay, ACKs (Section 4) and record received packets in a local eACK data structure (Section 5). This is representative of the effort required to integrate Packrat with any userspace transport protocol.

Low-latency media with FEC.. We emulate an audio streaming application that uses a simple repetition code for forward error correction. The server sends a packet every 20ms containing audio from the last 40ms (that is, data in packets overlap). Each audio packet contains 480 bytes of data, representing an audio stream at 96 Kbit/s. The client begins playback with 40ms in its buffer, stalling if frames arrive too late and fast-forwarding if behind the target buffer size. The client sends NACKs to indicate missing frames in its playback buffer and retransmits these NACKs (up to once per RTT) until it has received the missing frame. The server immediately retransmits data upon receiving a NACK. We measure one-way latency from the time the data is produced in the "real world" to when it is available in the client's buffer. For example, a frame containing 40 ms of data sent over a network path with 80 ms delay has a minimum one-way latency of 120 ms.

Reliable IP multicast stream. A single server streams 240-byte packets to a multicast IP address. Multiple clients subscribe to the multicast IP address. Clients send end-to-end NACKs to the multicast server to receive a unicast retransmission. We report the number of end-to-end (unicast) retransmissions, which captures both server and network load.

7 Evaluation

We evaluate our Packrat protocol implementation in an emulation study. We aim to answer the following questions:

	Data Sender (Server) ↔ Proxy	Data Receiver (Client) ↔ Proxy	Reorder Delay
"WiFi"	30ms, 20 Mb/s	2ms, 50 Mb/s, 4% loss	30ms
"Satellite"	50ms, 20 Mb/s	30ms, 50 Mb/s, 10% loss	65ms
"Cellular"	30ms, 20 Mb/s	10ms, 50 Mb/s, 10% loss	30ms

Table 3. The one-way delays, data rates, default loss values, and client-side ACK withholding delay (Section 4) for the settings we consider in our evaluations. All server-proxy links are lossless.

	eACK Frequency	Symbols	Cache Cap.
HTTP/3	10ms/16pkts	160	48 kB
Media	NACK	40	20 kB
Multicast	NACK	40	64 kB

Table 4. Packrat protocol configurations for each benchmark: client eACK transmission frequency, local eACK size, and maximum cache capacity.

(1) (Section 7.2) Does the Packrat protocol improve application goodput for a reliable download, latency for a media stream, and link overhead for a multicast application when compared to end-to-end retransmissions?
(2) (Section 7.3.2) Do the rateless and selective eACK optimizations described in Section 5.2 successfully reduce the link overheads of a Packrat connection?
(3) (Section 7.3.3, 7.3.4) What are the memory and CPU requirements of a Packrat proxy?

7.1 Methodology

We run emulation experiments in `mininet` using a linear, two-segment network topology with a server (transport sender), proxy, and client (transport receiver) or multicast clients. We parameterize each path segment in three dimensions: delay, bandwidth, and loss. We use `tc-netem` [40] to emulate loss. Experiments use random IID loss unless indicated otherwise; we also evaluate correlated (i.e., "bursty") loss using `tc-netem`'s implementation of the classical Gilbert-Elliott loss model, which models loss as a two-state Markov chain with "good" (low loss) and "bad" (high loss) states [18, 22, 40]. Egress interfaces use Random Early Detection `qdisc` [39] to model minimal congestive loss at the bottleneck link. Each segment is symmetric.

Table 3 describes the one-way delay, bottleneck data rate, and default loss value of each segment. We select far-path delays to approximate connections to a Wi-Fi access point, low-Earth orbit satellite ground station, and cellular base station. The sender-to-proxy segment—a cellular or wired network core—is reliable but high latency. The proxy-to-receiver segment—a wireless last mile—experiences non-congestive loss.

The reorder delay refers to the maximum duration for which the Packrat client may withhold end-to-end acknowledgments while awaiting an in-network retransmission; we configure this statically, though a Packrat client could adjust it dynamically as it estimates the client-to-proxy RTT. In selecting a reorder delay, we broadly aim to balance giving the proxy at least one chance to retransmit while maintaining timely end-to-end feedback. Table 4 describes the remaining Packrat protocol configurations for each benchmark. We select an eACK frequency to mirror the protocol's underlying acknowledgment scheme. The "maximum symbols" parameter refers to the number of symbols t in each local rIBLT as described in Table 1. We select a cache size that is small enough to evaluate optimistic eviction in Section 7.3.3.

Unless otherwise specified, the HTTP benchmark results are the median and IQRs of 20 trials. The media and multicast benchmark results are over 3 minutes. We define spurious retransmissions as the number of packets sent by the server containing duplicate frames. We measure link overheads by reading */sys/class/net/<iface>/statistics/** immediately before and after each benchmark. Cache memory usage is measured by logging cache updates and parsing the time-series logs. We use `rdtsc()` to measure CPU cycles in the proxy. We perform experiments on an AWS EC2 m4.xlarge instance with 16 GB of RAM and 4-CPU Intel Xeon E5 processor at 2.30 GHz. The instance runs Ubuntu 22.04.2 LTS with the Linux 6.80-1024-aws kernel.

In addition to benchmarking Packrat against end-to-end retransmissions, we aim to evaluate Packrat against a split connection. We implement a Picoquic connection splitter that decrypts and re-encrypts packets in two separate QUIC connections from one endpoint to another. Deploying this in reality would require a credentialed in-network middlebox.[4]

7.2 Application Performance

We find that in-network retransmissions via the Packrat protocol enable a variety of performance enhancements compared with end-to-end retransmissions—higher throughput, lower latency, and lower link overheads—for a variety of encrypted transport protocols when the data receiver is near a lossy path segment. In many cases, Packrat closely matches the performance of a split connection *without fate-sharing with endpoints.*

7.2.1 HTTP/3 file download. We measure a large HTTP/3 file download over our WiFi, cellular, and LEO satellite topologies and use Picoquic's implementations of both CUBIC and BBRv3 (Figure 5).

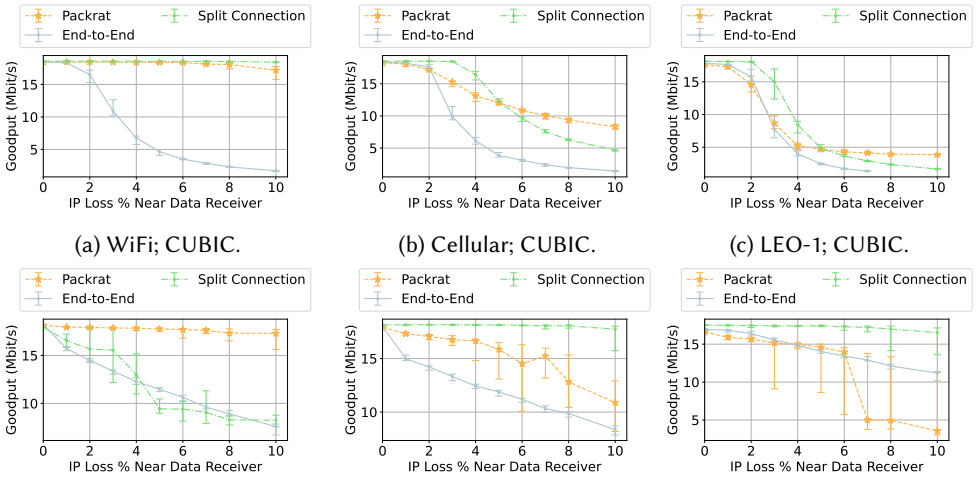
Packrat vs. End-to-End. The primary goal of Packrat is to provide protocol-agnostic, in-network assistance to an end-to-end connection without changing on-the-wire format or fate-sharing. Consistent with prior work, we observe that an end-to-end download over CUBIC performs poorly even with relatively low loss, and Packrat is able to recover dropped packets when the last-mile segment experiences relatively low latency. For example, with just 4% IP-layer loss in our emulated WiFi topology, Packrat improves goodput by 2.7× compared to end-to-end retransmissions over CUBIC and 1.4× over BBRv3. When the last-mile segment is high-latency, neither Packrat nor a split connection is able to mask significant loss from the transport sender over CUBIC.

In general, Packrat performance degrades as loss and latency increase. Under these conditions, the proxy may fail to successfully complete a retransmission before cache eviction, which can lead the client and proxy's local eACKs to become out-of-sync. In these cases, the proxy may fail to decode an eACK and thus must reset the Packrat connection, forcing the receiver to fall back to end-to-end retransmissions. In addition, as latency increases, the proxy has fewer "chances" to retransmit lost packets before the client sends an end-to-end acknowledgment, signaling loss to the server. Selectively withholding acknowledgments helps mitigate this, but there is a fundamental tradeoff: the client must balance timely end-to-end feedback with providing the proxy enough time to recover lost packets.

Interestingly, over high latency *and* high loss, an end-to-end Picoquic BBRv3 connection can match or outperform Packrat, which behaves erratically under high loss (Figure 5f). In this high-latency, satellite-like case, Packrat delays acknowledgments by 65ms and only has one opportunity to retransmit. If the Packrat connection is unable to recover a dropped packet on the first try, the subsequent delayed acknowledgment may (1) lead to a duplicate end-to-end acknowledgment and (2) inflate BBRv3's end-to-end RTT estimation. We note that the performance of BBRv3 differs by implementation [58], and Picoquic's in particular penalizes duplicate acknowledgments, which is not a fundamental feature of BBRv3.

Packrat vs. Split Connection, CUBIC. Over low-latency links (Section 7.2.1), Packrat nearly matches the performance of a split connection, suggesting that the congestion response to loss with Packrat can be both as good and as fair as QUIC with a connection splitter. When the last-mile segment has a higher latency (Section 7.2.1), Packrat sees higher goodput than the split connection with IP-layer loss above 6%. While the connection-splitting PEP can retransmit lost packets, it is still running CUBIC congestion control. While the proxy can quickly recover its congestion window

[4]MASQUE will eventually have a "forwarded mode" which does not require terminating connections in the proxy—however, this entails removing congestion control from the server-proxy path segment [49]



(a) WiFi; CUBIC. (b) Cellular; CUBIC. (c) LEO-1; CUBIC.

(d) WiFi; BBRv3. (e) Cellular; BBRv3. (f) LEO-1; BBRv3.

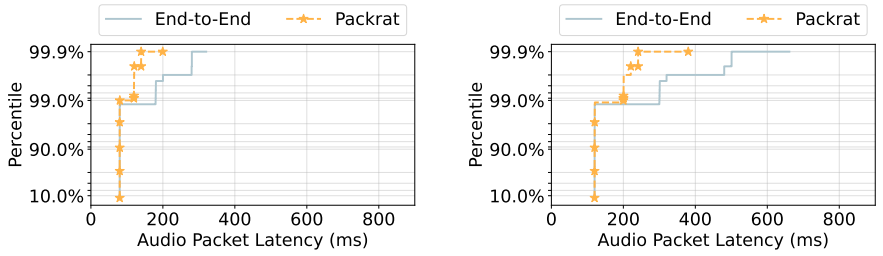
Fig. 5. Goodput of a large HTTP/3 file transfer using CUBIC and BBRv3 over the scenarios in Table 3 for a split, Packrat-assisted, and end-to-end connection. Packrat is able to recover in-network loss, outperforming end-to-end retransmissions when the latency between the client and proxy is relatively low, and typically approaching the goodput of a split connection.

following loss in the WiFi case (4ms RTT), it cannot in the cellular (20ms RTT) or satellite (60ms RTT) cases. In contrast, if a Packrat in-network retransmission is able to hide loss from the transport sender, the sender will not decrease its congestion window.

Packrat vs. Split Connection, BBRv3. Consistent with prior work [58], we observe that the connection-splitting PEP improves BBRv3 goodput even when the last-mile segment is high-latency (Section 7.2.1, 5f). In fact, a split BBRv3 connection achieves nearly full-throttle utilization over our high-latency link with 10% IP loss. Our split connection with BBRv3 performs poorly over an ultra-low-latency last-mile. BBRv3 controls transport sending rate in part with a delay estimation, which can behave poorly over low RTTs [2] (i.e., the user-space Picoquic implementation of BBRv3 is likely not built or tested to operate at a 4ms RTT granularity).

Bursty Loss. We assess the impact of loss burstiness on the performance of Packrat using the Gilbert-Elliott loss model supported in `netem`. To evaluate the impact of burstiness, we fix the overall loss rate and vary the expected time spent in "good" and "bad" states. We include a sample of these results in Section A for the WiFi and cellular network settings over CUBIC with 10% overall loss. For the same reasons the Packrat performance degrades over high loss, Packrat is not able to cope with bursty loss as effectively as a split connection. However, a Packrat assisted HTTP/3 download still exceeds the goodput of an end-to-end connection by at least 2× over WiFi and 1.5× over cellular (10% loss each).

7.2.2 Low-latency media with FEC. Packrat enables lower tail latency, which translates to fewer and shorter stalls, in a low-latency media stream over high-delay cellular and satellite network settings (Section 7.2.1). Recall that we measure one-way latency as the time between when data is encoded in the "real world" and when it is available in the client's buffer (Section 6.2). In these network settings, the minimum one-way latency of a packet is already high (e.g., 40 ms encoding + 80 ms one-way delay in the emulated satellite topology). For reference, 150 ms is the normal latency



(a) Cellular. (b) Satellite.

Fig. 6. CDF of audio packet latency for the low-latency media stream with FEC over two of the scenarios in Table 3. Latency represents the time from when the sender produces data to when the client can decode it. The Packrat-assisted stream achieves lower latency and fewer and shorter stalls than end-to-end retransmissions.

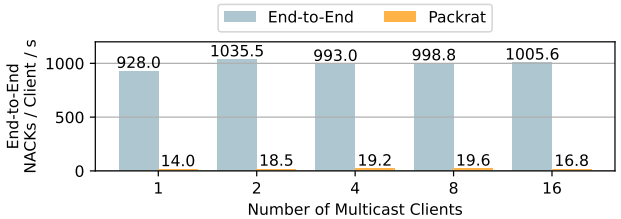


Fig. 7. Multicast benchmark over the cellular network setting (10% loss). Clients using Packrat send on average 59× fewer end-to-end NACKs than clients without Packrat. Other network scenarios see similar reductions in end-to-end retransmissions.

for VoIP, and anything above produces a noticeable drop in quality. Thus, network-generated retransmissions are crucial in lossy, high-latency settings.

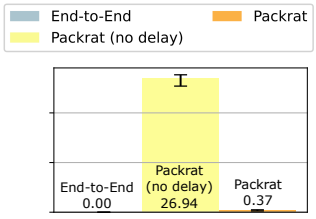
Both Packrat and end-to-end benefit from FEC, but Packrat enables shorter stalls when a frame still needs to be retransmitted. In this application, a frame needs to be retransmitted if two packets are lost in a row. In Figure 6b, at 10% loss, the 99th percentile delay of both mechanisms is the minimum of 190 ms. In the remaining percentile, one more out-of-order packet arrives after 20 ms, draining the last of the buffer and generating an eACK or NACK. The length of the stall is then the 60 ms RTT for a Packrat retransmission or the 240 ms RTT for an end-to-end retransmission.

7.2.3 Reliable IP multicast stream. Caching packets in the network like a CDN is inherently beneficial because it can reduce the congestion in the core network. This is one of the idealistic draws of IP multicast, but it is rarely used over the Internet in practice because a reliable stream may require an overwhelming number of unicast retransmissions from the server when there is loss. While CDNs and SFUs have helped with content distribution, these are not options for encrypted transport protocols without embedding trust in the network.

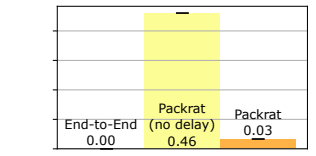
The Packrat proxy can handle in-network retransmissions for a large number of multicast clients. Each client using the Packrat protocol sends on average 16.8 end-to-end NACKs/second, a 59× reduction from 1005.6 without Packrat. This reduction scales with the number of clients. In-network retransmissions reduce both the load at the server and congestion in the network.

7.3 Communication, Memory, and CPU

7.3.1 Spurious Retransmissions and Selective ACK Withholding. We evaluate the impact of selectively withholding ACKs (Section 4) by comparing Packrat to a naive implementation without this

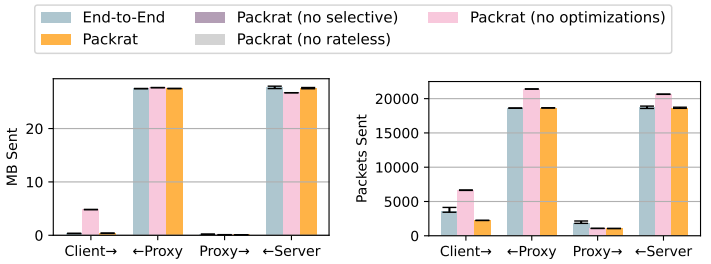


(a) HTTP/3, CUBIC, WiFi.

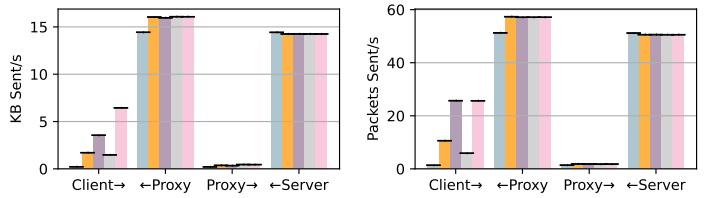


(b) Low-latency media, satellite.

Fig. 8. **Spurious Retransmissions.** A Packrat implementation that does not delay acknowledgment ranges (Section 4) triggers spurious retransmissions from the sender. Selectively delaying receiver-side acknowledgment ranges reduces them.



(a) HTTP/3 (bytes), CUBIC, WiFi. (b) HTTP/3 (packets), CUBIC, WiFi.



(c) Media (bytes), satellite. (d) Media (packets), satellite.

Fig. 9. **Link Overhead.** The number of bytes (left) and packets (right) sent between the client, proxy, and server. The link overheads from Packrat eACKs are comparable to those of the underlying acknowledgment scheme. Both clients send fewer bytes with rateless eACKs, and the media client sees fewer packets from selective eACKs (Section 5.2.2).

mechanism. Figure 8 shows that selectively withholding ACKs reduces the number of spurious retransmissions by 15× (Figure 8b) to 75× (Figure 8a), and Packrat successfully masks the majority of end-to-end loss from the data sender. We note that each retransmission contributes to link overhead and may trigger a congestion response.

7.3.2 Link Overheads. Figure 9 displays the number of packets and bytes sent on each link between the client, proxy, and server in the HTTP and media applications. As a proportion of the volume in bytes sent by both endpoints of the application, the HTTP client makes up 1.3% of the traffic with Packrat and 0.7% without. The media client makes up 8.2% of the traffic with Packrat and 1.0% without. We find that we can reasonably think of eACKs as control packets in the common case.

Rateless eACKs. Using the rateless property to send smaller eACKs (Section 5.2.2) reduces the number of bytes the client sends by 3.6× (Figure 9a, HTTP/3 over WiFi) and 1.8× (Figure 9c, media over satellite) compared to Packrat without the optimization. This property allows clients to configure the number of symbols less conservatively when initializing the Packrat connection while being able to dynamically adjust to changing loss conditions, particularly in less controlled non-emulation environments.

Sending eACKs on NACK. Sending an eACK only when the client would otherwise send a NACK reduces the number of packets the media client sends by 1.9× (Figure 9d). Note that without sending on NACK, we configured the media client to eACK every other packet to balance latency with frequent eACKing. The media client sends more packets than end-to-end because although it only ACKs when there is a missing *frame*, it eACKs when there is a missing *packet* because the FEC is opaque to the proxy. Because of its high throughput relative to loss, selectively eACKing does not impact the HTTP/3 download.

Base connection improvements. The client (data receiver) transmits a similar number of packets with and without Packrat (Figure 9b). Without Packrat, the client sends nearly twice as many end-to-end ACKs (we can derive this from the packets that the proxy forwards from the client to server). That is, Packrat actually lowers the network overhead between it and the data sender. The number of ACKs depends on the duration of the connection, and Packrat improves throughput such that the number of net packets (end-to-end ACKs and Packrat eACKs) sent by the client is similar with and without Packrat assistance. The IP multicast application further demonstrates how Packrat can reduce congestion closer to the server by caching retransmissions in the middle of the network (Section 7.2.1). This is especially impactful if the data is shared by multiple clients.

7.3.3 Memory Requirements. Like any in-network retransmission mechanism, the most significant memory usage at the proxy comes from the packet cache. Intuitively, this cache must store packets that the proxy has forwarded and has not yet received a client eACK for. We confirm that the cache utilization is equal to the packets forwarded during one client-proxy RTT plus expected loss, with occasional spikes when a client eACK is lost. The proxy optimistically evicts packets when the cache is full, keeping memory utilization bounded. We present measurement results in Section B.

7.3.4 CPU Requirements. We analyze the number of cycles that the proxy needs to process each packet and eACK in the CUBIC HTTP/3 benchmark over WiFi with 4% IP-layer loss. We use the per-packet overheads to extrapolate the capacity of a CPU-constrained proxy.

Each data packet on the base connection takes on average 1002 cycles to process, which is primarily attributable to adding the packet to the packet cache (a ring buffer). In the HTTP/3 benchmark over WiFi, each eACK takes the proxy on average 27716 cycles to process. This includes encoding a delta of packets forwarded since the last eACK, decoding the received eACK, updating the packet cache, and performing retransmissions. Encoding uses a total of $7156/27716 = 25.8\%$ and decoding $3560/27716 = 12.8\%$ of overall eACK processing.

In this application and network setting, we expect one eACK for every 16.3 data packets. If each data packet uses a full MTU of 1500 bytes, this means a single core of a 2.4 GHz CPU on a proxy would theoretically be able to handle 10.7 Gbit/s[5]. We note that this estimate does not account for the substantial CPU overhead involved in reading from and writing to the NIC—these can be significantly reduced using kernel bypass frameworks or specialized hardware.

8 Discussion

Evaluation Limitations. Our initial results suggest that (1) selectively withholding acknowledgment ranges at the transport layer in coordination with in-network retransmissions and (2) reconciling encrypted packet sets with the rIBLT are promising mechanisms for in-network retransmission. We recognize that evaluation with a single sender, in emulation, and in three network topologies is limited. We hope that our work motivates future exploration of such mechanisms in the wild and under contention. Additionally, future emulation studies would benefit from empirical characterization and models of modern wireless network loss patterns.

Implementation Limitations. Our prototype does not address the security of the initial handshake, though we note that an intercepted handshake would simply force the client to fall back on end-to-end retransmission (i.e., denial of the Packrat service). We implement our proxy as a network bridge (Section 6) to synchronize each epoch start; we suspect that future implementations could avoid this, such as through an initial short epoch of symmetric reconciliation. There are also opportunities

[5] 2.4 Gcycles/s $\times$ (16.3 data packets /(27716 $\cdot$ 1 + 1002 $\cdot$ 16.3) cycles) $\times$ (1500 bytes/data packet) $\times$ (8 bits/byte) = 10.7 Gbit/s.

to optimize our implementation, such as by applying kernel bypass networking at the proxy to reduce packet-processing latency.

Bidirectional Assistance. The Robin sidekick protocol [59] provides in-network acknowledgments, while Packrat provides in-network retransmissions. To assist bidirectional communication, a PEP and client could execute both protocols in parallel. Note that Packrat and Robin operate independently. The client and proxy must maintain separate set reconciliation data structures for each protocol—one for Robin, representing packets sent by the client, and one for Packrat, representing packets forwarded by the proxy. In Robin, the proxy sends quACKs; in Packrat, the client sends eACKs. As an optimization, Robin and Packrat could share the same sidekick connection, using an extra configuration step in which a client requests assistance through Robin, Packrat, or both depending on its knowledge of the base connection.

Ethics. All experiments are performed in emulation with locally generated traffic. We do not anticipate any ethical issues in this work.

9 Conclusion

In this paper, we describe and evaluate the Packrat protocol for sending network-originated retransmissions when the end-to-end transport is encrypted. We introduce an interface between the end-to-end transport protocol receiver and the adjacent Packrat connection to selectively withhold acknowledgment ranges to avoid spurious end-to-end retransmissions, and we use set-reconciliation techniques based on the Rateless IBLT to let the receiver efficiently acknowledge encrypted packets. Packrat can provide performance benefits in settings with high IP-layer loss without adding an additional layer of encapsulation or changing the end-to-end transport protocol.

Acknowledgments

We thank our shepherd, Lin Shihan; the anonymous reviewers; and the Stanford Systems and Networking Research Group for their valuable feedback and discussion. This work was supported in part by NSF grants 2045714, 2039070 and 2319080, DARPA contract HR001120C0107, the Sloan Research Fellowship, and by Google, Huawei, VMware, Dropbox, Amazon, and Meta Platforms. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the funding organizations.

References

[1] 3rd Generation Partnership Project (3GPP). 2025. *NR; NR and NG-RAN Overall Description; Stage 2.* Technical Report TS 38.300. 3GPP. https://www.3gpp.org/DynaReport/38300.htm Version 18.5.0.

[2] Akshita Abrol, Purnima Murali Mohan, and Tram Truong-Huu. 2024. FaiRTT: An Empirical Approach for Enhanced RTT Fairness and Bottleneck Throughput in BBR. In *2024 IEEE International Conference on Communications Workshops (ICC Workshops)*. IEEE, 673–678.

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

[4] Hari Balakrishnan, Venkata N Padmanabhan, Srinivasan Seshan, and Randy H Katz. 2002. A comparison of mechanisms for improving TCP performance over wireless links. *IEEE/ACM transactions on networking* 5, 6 (2002), 756–769.

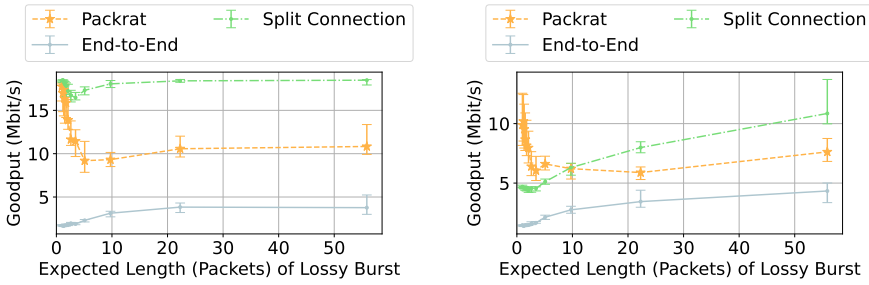
[5] Hari Balakrishnan, Srinivasan Seshan, Elan Amir, and Randy H. Katz. 1995. Improving TCP/IP performance over wireless networks. In *Proceedings of the 1st Annual International Conference on Mobile Computing and Networking* (Berkeley, California, USA) *(MobiCom '95)*. Association for Computing Machinery, New York, NY, USA, 2–11. doi:10.1145/215530.215544

[6] Burton H Bloom. 1970. Space/Time Trade-Offs in Hash Coding with Allowable Errors. *Commun. ACM* 13, 7 (1970), 422–426.

[7] John Border. 2020. Google QUIC over Satellite Links. Presentation, IETF PANRG interim. https://datatracker.ietf.org/meeting/interim-2020-panrg-01/materials/slides-interim-2020-panrg-01-sessa-google-quic-over-satellite-testing-update

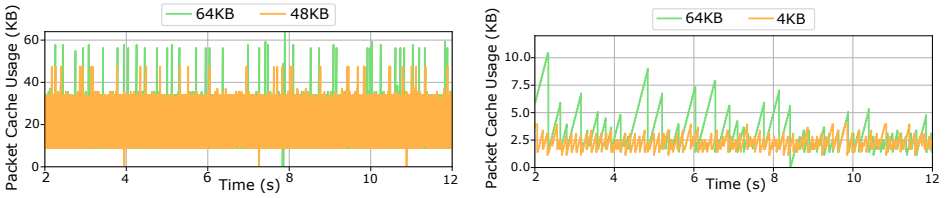
[8] John Border, Bhavit Shah, Chi-Jiun Su, and Rob Torres. 2020. Evaluating QUIC's Performance Against Performance Enhancing Proxy over Satellite Link. In *2020 IFIP Networking Conference (Networking)*. 755–760.
[9] John Border, Bhavit Shah, Chi-Jiun Su, and Rob Torres. 2020. Evaluating QUIC's Performance Against Performance Enhancing Proxy over Satellite Link. In *2020 IFIP Networking Conference (Networking)*. 755–760.
[10] 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. 2607–2611. doi:10.1109/VETECS.2006.1683339
[11] Neal Cardwell. 2024. BBRv3: Algorithm Overview and Google's Public Internet Deployment. Presentation, IETF 119 Brisbane, Congestion Control Working Group (CCWG). https://datatracker.ietf.org/meeting/119/materials/slides-119-ccwg-bbrv3-overview-and-google-deployment-00
[12] Neal Cardwell, Yuchung Cheng, C Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. 2017. BBR: Congestion-Based Congestion Control. *Commun. ACM* 60, 2 (2017), 58–66.
[13] Yuchung Cheng, Neal Cardwell, Nandita Dukkipati, and Priyaranjan Jha. 2021. The RACK-TLP Loss Detection Algorithm for TCP. RFC 8985. doi:10.17487/RFC8985
[14] Bryce Cronkite-Ratcliff, Aran Bergman, Shay Vargaftik, Madhusudhan Ravi, Nick McKeown, Ittai Abraham, and Isaac Keslassy. 2016. Virtualized Congestion Control. In *Proceedings of the 2016 ACM SIGCOMM Conference* (Florianopolis, Brazil) *(SIGCOMM '16)*. Association for Computing Machinery, New York, NY, USA, 230–243. doi:10.1145/2934872.2934889
[15] Jörg Deutschmann, Kai-Steffen Hielscher, and Reinhard German. 2023. CUBIC Local Loss Recovery vs. BBR on (Satellite) Internet Paths. In *2023 IEEE 29th International Symposium on Local and Metropolitan Area Networks (LANMAN)*. IEEE, 1–3.
[16] Mo Dong, Qingxi Li, Doron Zarchy, P Brighten Godfrey, and Michael Schapira. 2015. {PCC}: Re-architecting congestion control for consistent high performance. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. 395–408.
[17] Korian Edeline and Benoit Donnet. 2019. A Bottom-Up Investigation of the Transport-Layer Ossification. In *2019 Network Traffic Measurement and Analysis Conference (TMA)*. 169–176. doi:10.23919/TMA.2019.8784690
[18] Edwin O Elliott. 1963. Estimates of error rates for codes on burst-noise channels. *The Bell System Technical Journal* 42, 5 (1963), 1977–1997.
[19] David Eppstein and Michael T. Goodrich. 2011. Straggler Identification in Round-Trip Data Streams via Newton's Identities and Invertible Bloom Filters. *IEEE Trans. on Knowl. and Data Eng.* 23, 2 (Feb. 2011), 297–306. doi:10.1109/TKDE.2010.132
[20] Gorry Fairhurst and Lloyd Wood. 2002. Advice to link designers on link Automatic Repeat reQuest (ARQ). RFC 3366. doi:10.17487/RFC3366
[21] 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. doi:10.1007/978-3-642-32808-4_24
[22] Edgar N Gilbert. 1960. Capacity of a burst-noise channel. *Bell system technical journal* 39, 5 (1960), 1253–1265.
[23] Michael T Goodrich and Michael Mitzenmacher. 2011. Invertible bloom lookup tables. In *2011 49th Annual Allerton Conference on Communication, Control, and Computing (Allerton)*. IEEE, 792–799.
[24] Jim Griner, John Border, Markku Kojo, Zach D. Shelby, and Gabriel Montenegro. 2001. Performance Enhancing Proxies Intended to Mitigate Link-Related Degradations. RFC 3135. doi:10.17487/RFC3135
[25] Sangtae Ha, Injong Rhee, and Lisong Xu. 2008. CUBIC: a new TCP-friendly high-speed TCP variant. *SIGOPS Oper. Syst. Rev.* 42, 5 (July 2008), 64–74. doi:10.1145/1400097.1400105
[26] David A. Hayes, David Ros, and Özgü Alay. 2019. 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)*. 108–116. doi:10.1109/LCNSymposium47956.2019.9000661
[27] 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*. 181–194.
[28] Christian Huitema. 2025. picoquic. https://github.com/private-octopus/picoquic.
[29] IEEE Computer Society. 2005. IEEE Standard for Information technology–Local and metropolitan area networks–Specific requirements–Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications - Amendment 8: Medium Access Control (MAC) Quality of Service Enhancements.
[30] Jana Iyengar and Ian Swett. 2021. QUIC Loss Detection and Congestion Control. RFC 9002. https://www.rfc-editor.org/info/rfc9002
[31] Ankur Jain, Andreas Terzis, Hannu Flinck, Nurit Sprecher, Swaminathan Arunachalam, Kevin Smith, Vijay Devarapalli, and Roni Bar Yanai. 2017. *Mobile Throughput Guidance Inband Signaling Protocol*. Internet-Draft draft-flinck-mobile-throughput-guidance-04. Internet Engineering Task Force. https://datatracker.ietf.org/doc/draft-flinck-mobile-

throughput-guidance/04/ Work in Progress.
[32] Swastik Kopparty, Srikanth V.. Krishnamurthy, Michalis Faloutsos, and Satish K. Tripathi. 2002. Split TCP for Mobile Ad Hoc Networks. In *Global Telecommunications Conference, 2002. GLOBECOM '02. IEEE.* 138–142 vol.1.
[33] Mike Kosek, Hendrik Cech, Vaibhav Bajpai, and Jörg Ott. 2022. Exploring Proxying QUIC and HTTP/3 for Satellite Communication. In *2022 IFIP Networking Conference (IFIP Networking).* 1–9. doi:10.23919/IFIPNetworking55013.2022.9829773
[34] 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. doi:10.1109/MCOM.001.2000877
[35] Nicolas Kuhn, Emmanuel Lochin, Jérôme Lacan, Roksana Boreli, and Laurence Clarac. 2015. On the impact of link layer retransmission schemes on TCP over 4G satellite links. *International Journal of Satellite Communications and Networking* 33, 1 (2015), 19–42.
[36] Nicolas Kuhn, François Michel, Ludovic Thomas, Emmanuel Dubois, and Emmanuel Lochin. 2020. 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).* 1–7. doi:10.1109/ASMS/SPSC48805.2020.9268814
[37] Adam Langley, Alistair Riddoch, Alyssa Wilk, Antonio Vicente, Charles Krasic, Dan Zhang, Fan Yang, Fedor Kouranov, Ian Swett, Janardhan Iyengar, et al. 2017. The QUIC Transport Protocol: Design and Internet-Scale Deployment. In *Proceedings of the conference of the ACM special interest group on data communication.* 183–196.
[38] Hoang D Le and Anh T Pham. 2022. Link-layer retransmission-based error-control protocols in FSO communications: A survey. *IEEE Communications Surveys & Tutorials* 24, 3 (2022), 1602–1633.
[39] Linux Foundation. 2001. *tc-red.* https://man7.org/linux/man-pages/man8/tc-red.8.html Linux man page for the Random Early Detection option for tc-qdisc.
[40] Linux Foundation. 2011. *tc-netem.* https://man7.org/linux/man-pages/man8/tc-netem.8.html Linux man page for tc-netem.
[41] Aitor Martin and Naeem Khademi. 2022. On the Suitability of BBR Congestion Control for QUIC Over GEO SATCOM Networks. In *Proceedings of the Workshop on Applied Networking Research.* 1–8.
[42] Matthew Mathis, Jeffrey Semke, Jamshid Mahdavi, and Teunis Ott. 1997. The Macroscopic Behavior of the TCP Congestion Avoidance Algorithm. *ACM SIGCOMM Computer Communication Review* 27, 3 (1997), 67–82.
[43] Attila Mihály, Szilveszter Nádas, Sándor Molnár, Zsolt Krämer, Robert Skog, and Marcus Ihlar. 2017. Supporting multi-domain congestion control by a lightweight PEP. In *2017 International Conference on Internet of Things, Embedded Systems and Communications (IINTEC).* 105–110. doi:10.1109/IINTEC.2017.8325922
[44] Yaron Minsky, Ari Trachtenberg, and Richard Zippel. 2003. Set reconciliation with nearly optimal communication complexity. *IEEE Transactions on Information Theory* 49, 9 (2003), 2213–2218. doi:10.1109/TIT.2003.815784
[45] Ayush Mishra, Sherman Lim, and Ben Leong. 2022. Understanding speciation in QUIC congestion control. In *Proceedings of the 22nd ACM Internet Measurement Conference.* 560–566.
[46] Surendra Pandey, Raja Moses Manoj Kumar Eda, and Debabrata Das. 2020. Efficient Reordering-Reassembly PDCP and RLC Window Management Algorithm in 5G and Beyond. In *2020 IEEE International Conference on Electronics, Computing and Communication Technologies (CONECCT).* 1–6. doi:10.1109/CONECCT50063.2020.9198680
[47] Giorgos Papastergiou, Gorry Fairhurst, David Ros, Anna Brunstrom, Karl-Johan Grinnemo, Per Hurtig, Naeem Khademi, Michael Tüxen, Michael Welzl, Dragana Damjanovic, and Simone Mangiante. 2017. De-Ossifying the Internet Transport Layer: A Survey and Future Perspectives. *IEEE Communications Surveys & Tutorials* 19, 1 (2017), 619–639. doi:10.1109/COMST.2016.2626780
[48] Abhinav Pathak, Angela Wang, Cheng Huang, Albert Greenberg, Y. Charlie Hu, Randy Kern, Jin Li, and Keith Ross. 2010. Measuring and Evaluating TCP Splitting for Cloud Services.. In *Proceedings of the 11th International Conference on Passive and Active Measurement.* 41–50.
[49] Tommy Pauly, Eric Rosenberg, and David Schinazi. 2025. *QUIC-Aware Proxying Using HTTP.* Internet-Draft draft-ietf-masque-quic-proxy-07. Internet Engineering Task Force. https://datatracker.ietf.org/doc/draft-ietf-masque-quic-proxy/07/ Work in Progress.
[50] 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. doi:10.1109/ACSSC.2017.8335489
[51] J. Saltzer, D. Reed, and D. Clark. 1984. End-to-end Arguments in System Design. *TOCS* 2 (Nov. 1984), 277–288.
[52] David Schinazi. 2022. Proxying UDP in HTTP. RFC 9298. doi:10.17487/RFC9298
[53] David Schinazi. 2025. *The MASQUE Proxy.* Internet-Draft draft-schinazi-masque-proxy-05. Internet Engineering Task Force. https://datatracker.ietf.org/doc/draft-schinazi-masque-proxy/05/ Work in Progress.
[54] Ludovic Thomas, Emmanuel Dubois, Nicolas Kuhn, and Emmanuel Lochin. 2019. Google QUIC performance over a public SATCOM access. *International Journal of Satellite Communications and Networking* 37, 6 (2019), 601–611.



(a) "WiFi"; CUBIC; 10% loss. (b) "Cellular"; CUBIC; 10% loss.

Fig. 10. Goodput under Gilbert-Elliott loss model, fixing overall loss rate and varying time spent in good/bad states. While the split connection is better able to manage bursty loss, Packrat maintains at least 2× goodput over end-to-end retransmissions.



(a) HTTP/3 file download. (b) Low-latency media with FEC.

Fig. 11. The number of bytes in the cache over a 10-second period with both bounded (48 KB or 4 KB) and unbounded (64 KB) cache sizes. The proxy uses optimistic eviction if an incoming packet causes the cache to exceed its capacity.

[55] Felix Weinrank, Michael Tüxen, and Erwin P Rathgeb. 2020. RACK for SCTP. In *2020 IEEE 28th International Conference on Network Protocols (ICNP)*. IEEE, 1–6.

[56] Greg White, Ingemar Johansson, Dibakar Das, and Chris Box. 2025. *Proposal for Updates to Guidance on Packet Reordering*. Internet-Draft draft-white-intarea-reordering-02. Internet Engineering Task Force. https://datatracker.ietf.org/doc/draft-white-intarea-reordering/02/ Work in Progress.

[57] Lei Yang, Yossi Gilad, and Mohammad Alizadeh. 2024. Practical rateless set reconciliation. In *Proceedings of the ACM SIGCOMM 2024 Conference*. 595–612.

[58] Gina Yuan, Thea Rossman, and Keith Winstein. 2025. Internet Connection Splitting:{What's} Old is New Again. In *2025 USENIX Annual Technical Conference (USENIX ATC 25)*. 867–887.

[59] Gina Yuan, Matthew Sotoudeh, David K Zhang, Michael Welzl, David Mazières, and Keith Winstein. 2024. Sidekick:In-Network Assistance for Secure End-to-End Transport Protocols. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. 1813–1830.

[60] Gina Yuan, David K Zhang, Matthew Sotoudeh, Michael Welzl, and Keith Winstein. 2022. Sidecar: In-Network Performance Enhancements in the Age of Paranoid Transport Protocols. In *Proceedings of the 21st ACM Workshop on Hot Topics in Networks*. 221–227.

A Loss Bursts

We evaluate whether Packrat can cope with bursty loss using the Gilbert-Elliott two-state Markov chain model supported in `tc netem` [40]. The results pictured fix loss at 10% and vary the length of "good" and "bad" states, which are set to have 2% loss and 80% loss, respectively. While the split connection is better able to manage bursty loss, Packrat maintains at least 2× goodput over end-to-end retransmissions. Packrat is more significantly impacted by loss at lower latency.

B Cache Capacity Measurements

Like any in-network retransmission mechanism, the most significant memory usage at the proxy comes from the packet cache. Figure 11 shows the number of bytes in the packet contents of the cache over a 10-second period of (Figure 11a) the HTTP/3 file download over WiFi and (Figure 11b) the low-latency media stream over satellite.

HTTP/3 Download. The minimum cache size is ≈10 kB, which is approximately what we expect for a single eACK interval between the client and proxy. Spikes occur when an eACK is dropped. The proxy resets the connection if there is an error (e.g., if it optimistically evicted a necessary retransmission), which temporarily resets the cache size to zero.

Low-latency media with FEC. The media application relies on optimistic eviction to keep the cache small, since the client selectively eACKs only when it receives a NACK (Section 6.2). Because the low-latency media stream is also lower throughput (i.e., low outstanding packets per proxy-client RTT), we can leverage a smaller cache size.

Reliable IP multicast stream. Our multicast application has a fixed packet cache size of 64 kB. When there are 16 clients, we find that the proxy uses, on average, 0.178 KB across all virtual caches for a very low overhead of 12 bytes/client. In general, the overheads of the virtual caches are proportional to the number of outstanding retransmissions.

Received December 2025; accepted April 2026