# Striped Tunnels: Reliable Transport from Unreliable Paths

Vidur Gupta, Carlos Hernandez-Meza, Sebastian Fisher, Aditya Sriram
Stanford University
{vidur, carloshz, sbfisher, adisram}@stanford.edu

## Abstract

Multipath transport protocols face a tradeoff between fault tolerance and efficiency. Striping data across paths maximizes throughput but provides no redundancy, while replicating data tolerates failures but at multiplicative bandwidth cost. We present striped tunnels, a transport system that uses (k,n) erasure coding to achieve fault tolerance and bandwidth aggregation simultaneously. The sender encodes each block into n shares distributed across independent paths, and the receiver reconstructs from the first k to arrive, so that path failures require no retransmission. On an ns3 emulation with per-path packet loss, coded QUIC with a coding-aware congestion controller achieves up to an 11× throughput advantage over vanilla QUIC at 5% loss while remaining TCP-friendly (Jain fairness index 0.998).

## 1 Introduction

Many real networks contain multiple unique links between two endpoints, such as multi-homed hosts, SD-WAN overlays, and data center fabrics with ECMP. These topologies expose an interesting tradeoff for transporting network data. We ask whether multiple paths can be used to increase throughput while simultaneously adding redundancy to introduce fault tolerance.

We draw inspiration from RAID [14]. We define *striping* analogously to RAID-0: split data across multiple paths, aggregating their bandwidths but susceptible to a single failure. A *replicated* design, analogous to RAID-1, sends full copies on each path, wasting bandwidth but achieving high fault tolerance. *Erasure coding* offers the analogue of parity-based redundancy. Assuming $n$ links exist, we code $k$ data shares into $n$ shares such that only $k$ suffice to reconstruct the data, tolerating $n-k$ path losses at an overhead of only $n/k\times$ rather than $n\times$ bandwidth.

In this paper, we present *Striped Tunnels*, a user space layer beneath QUIC [6] that uses erasure codes to improve reliability in lossy multipath networks. QUIC datagrams are partitioned into coding generations, erasure coded, and sent across multiple paths. The receiver reconstructs a generation as soon as any $k$ shares arrive, so path failures and losses can be absorbed without retransmission or handoff within QUIC itself. This makes the mechanism topology-agnostic. The multiple paths may come from multiple interfaces, overlay relays, or independent WAN links.

The QUIC setting also changes the motivation relative to some prior FEC systems. Rather than primarily targeting low-latency recovery on a single path, as in some QUIC-FEC proposals [12, 13], our focus is resilient *multipath* transmission, sustaining useful throughput when paths differ in loss, bandwidth, or failure behavior. This framing makes the RAID analogy particularly apt. In storage, parity protects against disk failures while preserving much of the throughput benefit of striping. In transport, coded striping protects against path failures while preserving much of the throughput benefit of parallel paths.

The idea of coding across multiple paths is not new [2, 17]. Our contribution is showing that this approach works cleanly beneath QUIC with a TCP-friendly congestion controller, and quantifying the tradeoffs in a realistic emulation. We implement CodingAwareReno, a coding-aware congestion controller that satisfies RFC 9265 [7] through wire-byte accounting, and evaluate it under ns3 emulation where coded QUIC sustains 11× the throughput of vanilla QUIC at 5% per-path

loss.

## 2 Background

**Multipath Routing.** Certain real-world contexts have network topologies in which multiple paths exist between endpoints, including multi-homed hosts with connections to several ISPs, SD-WAN overlays that tunnel traffic across independent WAN links, and data center fabrics using ECMP over Clos topologies. Multipath TCP (MPTCP) [3, 4] extends TCP by maintaining independent subflows per path and aggregating their bandwidth at the receiver. However, because reliability is handled at the subflow level, losses or failures on one path must be repaired through retransmission or scheduler adaptation. MPLOT [17] combines multipath transport with erasure coding so that losses on individual paths can be recovered without retransmission, while FMTCP [2] explores a similar direction using fountain codes. Our system operates entirely in user space above UDP, requiring no kernel or protocol modifications.

**(k, n) Erasure Coding.** Erasure coding generalizes replication by producing $n$ coded shares from $k$ original data shards such that any $k$ of the $n$ shares suffice to recover the original data. We use a systematic Reed-Solomon code [16] over $\mathrm{GF}(2^8)$. Encoding is defined by an $n \times k$ Vandermonde matrix $\mathbf{V}$ with entries $V_{ij} = \alpha_i^j$ for distinct nonzero elements $\alpha_i \in \mathrm{GF}(2^8)$. Given a data block $\mathbf{d} \in \mathrm{GF}(2^8)^k$, the $n$ shares are computed as $\mathbf{s} = \mathbf{V}\mathbf{d}$. The Vandermonde construction guarantees that every $k \times k$ submatrix of $\mathbf{V}$ is invertible [11], so any $k$ received shares can recover $\mathbf{d}$ via Gauss-Jordan elimination over $\mathrm{GF}(2^8)$. A code with parameters $(k, n)$ tolerates up to $n - k$ erasures while incurring a bandwidth overhead factor of $n/k$.

## 3 Design and Model

Our *striped tunnels* system inserts an erasure coding layer between QUIC and UDP (Figure 1). Because QUIC already runs in user space over UDP, this placement requires no kernel modifications.

We can model our striped tunnels approach with a simple multipath network topology. Refer to Figure 2 for such a topology, where two hosts are connected by $n$ network links labeled $\ell_1$ through $\ell_n$.

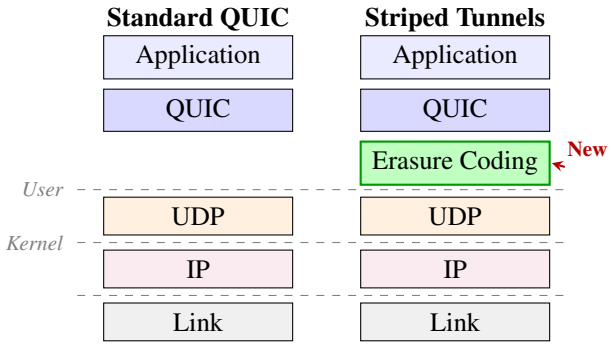


Figure 1: Network stack placement. QUIC operates at the application layer above UDP. Striped tunnels inserts an erasure coding layer between QUIC and UDP, requiring no kernel modifications.

On the send side, outgoing QUIC datagrams are buffered by the coding layer until $k$ have accumulated, forming one coding group. The $k$ datagrams are concatenated and erasure-coded into $n$ coded shares. Each share is wrapped in a 12-byte header carrying the group index, share index, coding parameters, and per-datagram lengths. All $n$ shares are sent to the same receiver address; the sender requires no knowledge of the underlying path topology. In our ns3 evaluation, gateway gw_A uses random ECMP to distribute shares across $n$ parallel links toward gw_B (Figure 2).

On the receive side, the coding layer demultiplexes incoming packets by group index. As soon as any $k$ distinct shares arrive for a group, the receiver inverts the corresponding $k \times k$ Vandermonde submatrix over $\mathrm{GF}(2^8)$ and recovers the original datagrams. Path failures are invisible to QUIC, and up to $n-k$ paths may fail per group without data loss or retransmission.

**Bandwidth Overhead Comparison.** A replication strategy transmits $n$ full copies of each packet, one per link, requiring $n\times$ base bandwidth. A striping approach transmits $n$ unique shares, requiring only $1\times$ bandwidth but with zero fault tolerance. Our erasure coded approach transmits $n$ coded shares at $n/k\times$ bandwidth.

**Throughput model.** If each path has capacity $r$, replication achieves throughput $r$ (only one copy matters). Striping achieves $n \times r$. Coding achieves $k \times r$, since only the first $k$ shares count.

**Fault Tolerance.** Replication across $n$ paths yields $(n-1)$-fault tolerance. Striping has zero fault tolerance. Erasure coding achieves $(n-k)$-fault tolerance.

**Reliability under loss.** If each share is lost indepen-

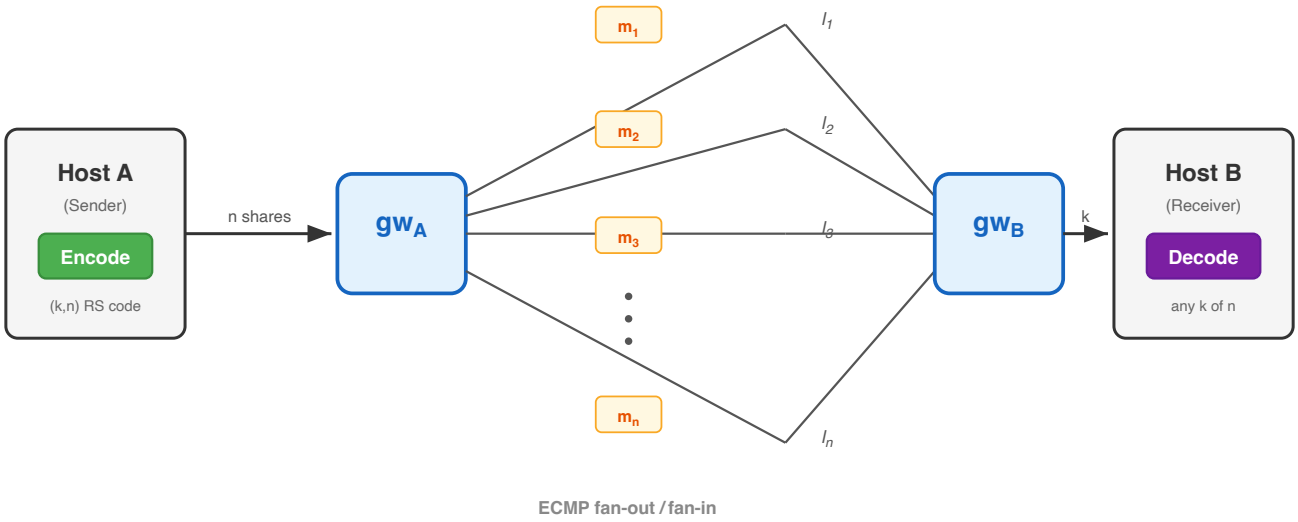


Figure 2: Gateway fan-out topology. Host A encodes packet $p$ into $n$ coded shares $m_1, \ldots, m_n$ via $(k,n)$ Reed-Solomon coding. Shares are forwarded to gateway gw_A, which distributes them across $n$ parallel links $\ell_1, \ldots, \ell_n$ via ECMP. Gateway gw_B forwards arriving shares to Host B, which decodes the original packet from any $k$ of the $n$ shares.

dently with probability $p$, the group failure probability is $P_{\text{fail}} = \sum_{i=n-k+1}^{n} \binom{n}{i} p^i (1-p)^{n-i}$. For a file of $B$ blocks, the transfer succeeds with probability $(1-P_{\text{fail}})^B$. At 2% per-path loss with $(k{=}3, n{=}5)$, $P_{\text{fail}} \approx 0.004\%$, so a 1600-block transfer succeeds with probability 94%. Increasing $n$ to 7 drops $P_{\text{fail}}$ below $10^{-6}$.

## 4 Implementation and Evaluation

### 4.1 Implementation

We implement the striped tunnels coding layer in Python beneath aioquic [10], a Python QUIC library. Full details of the sender and receiver data paths appear in Appendix A. Outgoing QUIC datagrams are buffered until $k$ have accumulated to form one coding group. The group is erasure-coded into $n$ shares, each wrapped in a striped tunnels packet (12+2k bytes fixed header). All $n$ shares are sent to a single receiver address; an intermediate gateway (gw_A in our ns3 topology) uses random ECMP to spread shares across $n$ parallel links toward the receiver-side gateway (gw_B). On receipt, the coding layer recovers the group as soon as any $k$ shares arrive and injects the reconstructed datagrams into the QUIC stack, so that QUIC's loss detector never sees them.

We also implement a coding-aware congestion controller (`CodingAwareReno`) that accounts for the $n/k$ bandwidth expansion of the coding layer. The controller uses standard TCP Reno AIMD parameters (0.5× multiplicative decrease, 1 MSS/RTT additive increase, 10×MSS initial window) but inflates all byte accounting by $n/k$. This wire-byte accounting ensures that the aggregate traffic on each link does not exceed what standard Reno would produce, satisfying RFC 9265 [7]. From QUIC's perspective, the coding layer is analogous to link-layer FEC, reducing the effective loss rate without circumventing congestion signals. One limitation of this approach is that the coding layer does not distinguish random loss from congestion-induced loss. If shares are dropped due to congestion, the coding layer may absorb them before QUIC can react. The wire-byte inflation partially compensates, since the coded flow is already constrained to $1/(n/k)$ of the raw sending rate, but a tighter integration with per-path congestion signals remains future work.

**Emulation environment.** QUIC experiments run in ns3-3.47 real-time emulation with Python bindings (cppyy). The gateway fan-out topology (sender → gw_A → five 50 Mbps/10 ms middle links → gw_B → receiver) uses random ECMP on gw_A and configurable forward-

| | Standard Reno | | CodingAwareReno | |
|---|---|---|---|---|
| **Loss** | **Vanilla** | **Coded** | **Coded** | **vs. Vanilla** |
| 0% | 46.5 | 27.7 | 27.1 Mbps | 0.6× |
| 2% | 2.7 | 26.9 | 23.8 Mbps | 8.8× |
| 5% | 1.5 | 19.5 | 16.5 Mbps | 11.0× |

Table 1: QUIC throughput (Mbps) on a 50 MB transfer in ns3 gateway topology (5 paths, 50 Mbps each, 10 ms one-way delay) at varying per-path loss rates. Vanilla QUIC uses standard NewReno. Coded QUIC uses ($k$=3, $n$=5) coding; "CodingAwareReno" additionally uses the coding-aware congestion controller with wire-byte accounting.

only loss on the middle links. Real applications (aioquic) run in Linux network namespaces connected to ns3 via TapBridge in UseBridge mode. We also run direct-UDP experiments over Linux network namespaces with per-path bandwidth and loss via `tc netem`. All file transfers are SHA256-verified end-to-end.

## 4.2 Coded QUIC under Loss

Table 1 reports QUIC throughput across a per-path loss sweep in the ns3 gateway topology (5 paths, 50 Mbps each, 10 ms one-way delay, ≈24 ms RTT).

**Zero loss.** Both vanilla and coded QUIC use a single logical connection; ECMP distributes packets across the five middle links, but QUIC's congestion controller sees one aggregate path. Vanilla QUIC achieves 46.5 Mbps, close to the single-path ceiling. Coded QUIC achieves only 27.7 Mbps because the $n/k$=1.67× coding overhead inflates wire bytes without a corresponding gain in QUIC-visible goodput. (The direct-UDP and MPTCP experiments in Sections 4.3 and 4.5 use per-path sockets and achieve higher aggregate throughput.)

**Under loss.** Vanilla QUIC collapses rapidly because its NewReno congestion controller interprets every lost packet as a congestion signal and halves its window, reducing throughput to 2.7 Mbps at 2% loss and 1.5 Mbps at 5% loss. Standard TCP throughput analysis predicts throughput $\propto 1/\sqrt{p}$, consistent with our measurements. Coded QUIC with standard Reno maintains 26.9 Mbps and 19.5 Mbps respectively, achieving 10× and 13× speedups, because the coding layer absorbs individual share losses before they reach QUIC's loss detector.

**CodingAwareReno.** The TCP-friendly CodingAwareReno controller uses standard Reno AIMD parameters but inflates byte accounting by $n/k$, so that the congestion window limits *wire* throughput rather than application throughput. This reduces coded QUIC performance slightly (23.8 Mbps at 2% loss, 16.5 Mbps at 5%) relative to standard Reno with coding, because the inflated byte count causes earlier window saturation. The tradeoff is TCP-friendliness. In a fairness test, a TCP flow sharing the bottleneck with CodingAwareReno-coded QUIC achieved 1.38 Mbps, compared to 1.50 Mbps when running alone (an 8% reduction, Jain fairness index 0.998). Even with the fairness penalty, coded QUIC under CodingAwareReno sustains an 8.8× to 11× advantage over vanilla QUIC at 2–5% loss rates.

**Severe loss.** At 10% per-path loss, coded QUIC with standard Reno still delivers 6.6 Mbps versus vanilla's 1.0 Mbps (6.6×), though both are significantly degraded. At this loss rate, the coding layer itself begins to fail frequently, since more than $n-k$=2 of 5 shares are lost per group with nontrivial probability.

**Zero-loss overhead.** The 0.6× penalty at zero loss is the cost of always-on redundancy, analogous to write amplification in RAID-5 storage. Each datagram generates $n/k = 1.67\times$ wire traffic, and wire-byte accounting constrains the congestion window proportionally. At nonzero loss the tradeoff reverses. Vanilla QUIC halves its window on every lost packet, while the coding layer absorbs most losses before they reach QUIC's loss detector, producing far fewer congestion events.

**Methodology.** Each row in Table 1 represents a single 50 MB transfer in ns3 real-time emulation with aioquic running in Linux namespaces connected via TapBridge to the ns3 topology. All transfers are SHA256-verified end-to-end.

## 4.3 Coded vs. Replicated vs. Striped

Using direct-UDP over Linux network namespaces (five 50 Mbps paths with per-path sockets), Figure 3 compares receiver throughput across three multipath strategies on a 100 MB transfer ($k$=3, $n$=5, 32 KB blocks) under three fault schedules: no failure, one path killed at the 50% mark, and two paths killed at the 50% mark.

The striped baseline splits the file into $n$ equal chunks with no redundancy. It achieved full throughput only when all paths were healthy; in both fault conditions it timed out, illustrating zero fault tolerance. The replicated baseline sends a full copy on every path. It completed

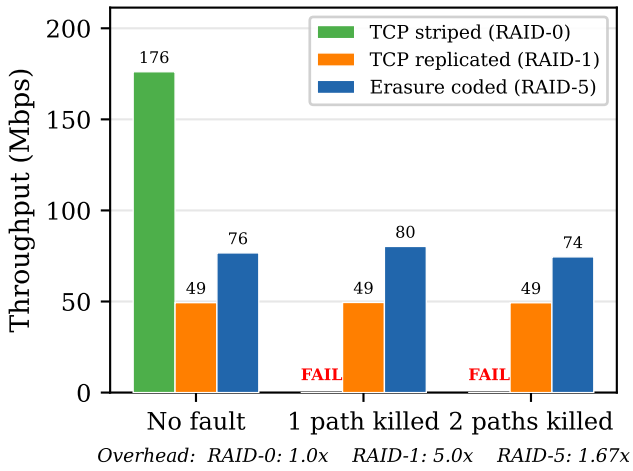


Figure 3: Receiver throughput for coded ($k$=3, $n$=5), TCP-replicated, and TCP-parallel (striped) baselines on a 100 MB transfer over five 50 Mbps paths under three fault schedules. TCP-parallel timed out in all fault conditions (30 s limit); coded throughput is stable across all three schedules.

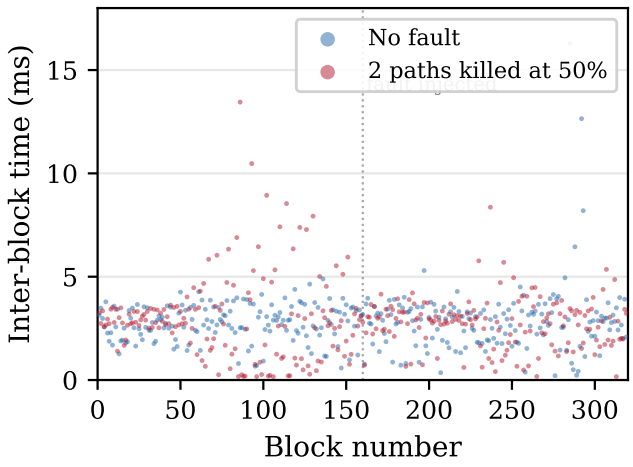


Figure 4: Per-block inter-arrival time over block number for coded ($k$=3, $n$=5) under two fault schedules: no fault (blue) and two paths killed at the 50% mark (red). The vertical line marks the fault injection point. No timing discontinuity is visible at the failure point, confirming disruption-free delivery.

in all three schedules at ≈49 Mbps, confirming fault tolerance but consuming $n\times$ bandwidth. The ($k$=3, $n$=5) coded transfer completed at 76 Mbps (no-fault), 80 Mbps (one-path failure), and 74 Mbps (two-path failure), consistently ≈55% higher than the replicated baseline. Up to $n-k$=2 simultaneous path failures are masked without retransmission.

These experiments use application-level pacing rather than congestion control, so the throughput numbers should be compared relative to each other rather than taken as achievable rates on the open Internet. The congestion-controlled evaluation appears in Section 4.2.

### 4.4 Disruption-Free Delivery

Using the same direct-UDP namespace testbed, Figure 4 shows per-block inter-arrival time over block number for a 10 MB coded transfer under two representative fault schedules. The no-fault case and the two-paths-killed case are overlaid, with the fault injection point marked by a vertical line. No discontinuity or spike is visible at the failure point, confirming that path failures are invisible at the receiver. Receiver throughput varies by less than 5% across five fault schedules (no fault, kill 1 at 25%, kill 1 at 50%, kill 2 at 50%, sequential kills), ranging from 91.5 to 96.1 Mbps.

### 4.5 MPTCP Comparison

We compare our direct-UDP coded transport against kernel MPTCP on EC2 (five paths, 50 MB transfer). The coded side uses application-level pacing rather than congestion control, so absolute throughput values are not directly comparable; the relevant observation is the contrasting failure behavior.

When all paths are healthy, MPTCP achieves 228.9 Mbps versus coded's 140.0 Mbps. Killing one path reduces MPTCP to 119.4 Mbps (−48%) while coded throughput is unchanged at 140.7 Mbps. Killing two paths drops MPTCP to 87.8 Mbps (−62%) while coded transport remains at 140.5 Mbps (Figure 5). The coded system's throughput is constant up to the fault tolerance limit $n-k$, while MPTCP must detect the failure, retransmit lost segments, and redistribute load.

## 5 Related Work

**Coded multipath transport.** MPLOT [17] pioneered erasure coding with multipath transport over wireless mesh networks. FMTCP [2] applied fountain codes across MPTCP subflows in simulation. Song et al. [18] studied erasure codes within MPTCP, as did earlier work on systematic coding for MPTCP subflows.

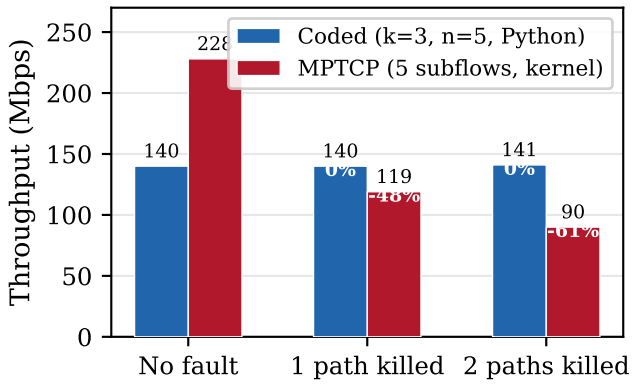


Figure 5: Coded ($k$=3, $n$=5) vs. kernel MPTCP under path failures.

TCP/NC [19] inserted random linear network codes below TCP on a single path. Our system differs in operating entirely in user space above UDP, requiring no kernel changes.

**FEC in QUIC and congestion control.** QUIC-FEC [13] and FlEC [12] integrated FEC into QUIC for single-path latency optimization. RFC 9265 [7] establishes that FEC should not hide congestion signals. Tetrys [9] provides on-the-fly network coding within individual flows. Our CodingAwareReno satisfies RFC 9265 through wire-byte accounting. Barakat and Altman [1] analyzed the bandwidth tradeoff between TCP and link-level FEC.

**Network coding and multipath TCP.** Linear network coding [5, 8] showed coding at intermediate nodes can achieve multicast capacity. CodedBulk [20] applied this to inter-datacenter transfers but requires SDN control. Our system codes only at endpoints. MPTCP [3, 4] with coupled congestion control [15] can replicate traffic across subflows for reliability, but at $n\times$ bandwidth cost. Our system provides fault tolerance at $n/k$ overhead while aggregating $k$ paths.

## 6 Discussion and Conclusion

We have shown that erasure-coded multipath transport provides reliable delivery from unreliable paths at tunable bandwidth cost. On ns3 emulation, coded QUIC with CodingAwareReno achieves up to 11$\times$ throughput over vanilla QUIC at 5% loss while remaining TCP-friendly (Jain index 0.998). In controlled experiments without congestion control, coded transport maintains constant throughput through path failures that degrade both MPTCP and striped baselines, while achieving fault tolerance at $n/k$ overhead rather than the $n\times$ cost of replication.

**Congestion control.** Our direct-UDP variant uses application-level pacing, which is acceptable in controlled emulation but not suitable for the open Internet [7]. Our coded QUIC variant addresses this with CodingAwareReno, which inflates byte accounting by $n/k$ to ensure TCP-fair behavior. An open question is how to adapt MPTCP's coupled congestion control [15] to coded shares, where a lost share may be absorbed by parity rather than retransmitted.

**Adaptive redundancy.** Fixed $(k,n)$ wastes bandwidth when paths are healthy and provides insufficient protection when loss is high. An adaptive scheme would adjust $n$ dynamically, starting with $n$=$k$ (zero overhead) and increasing when loss is detected. The most promising direction combines MPTCP with coding, using MPTCP normally and activating the coded layer when persistent path failures are detected.

## Acknowledgments

We thank David Mazières for mentoring us throughout this project and meeting with us regularly, Keith Winstein for early guidance that helped point us in the right direction, and Akshay Srivatsan for his feedback and encouragement.

## 7 AI Usage

We used AI tools for assistance with the implementation of our design, however we designed the protocol and experiments ourselves.

## References

[1] Chadi Barakat and Eitan Altman. Bandwidth tradeoff between tcp and link-level fec. *Computer Networks*, 39(2):133–150, 2002.

[2] Yong Cui, Leilei Wang, Xin Wang, Hao Wang, and Yining Wang. Fmtcp: A fountain code-based multipath transmission control protocol. *IEEE/ACM Transactions on Networking*, 23(4):1207–1220, 2015.

[3] Alan Ford, Costin Raiciu, Mark Handley, and Olivier Bonaventure. Tcp extensions for multipath operation with multiple addresses. RFC 6824, 2013.

[4] Alan Ford, Costin Raiciu, Mark Handley, Olivier Bonaventure, and Christoph Paasch. Tcp extensions for multipath operation with multiple addresses. RFC 8684, 2020.

[5] Tracey Ho, Muriel Médard, Ralf Koetter, David Karger, Michelle Effros, Jun Shi, and Ben Leong. A random linear network coding approach to multicast. *IEEE Transactions on Information Theory*, 52(10):4413–4430, 2006.

[6] Jana Iyengar and Martin Thomson. Quic: A udp-based multiplexed and secure transport. RFC 9000, 2021.

[7] Nicolas Kuhn, Emmanuel Lochin, François Michel, and Michael Welzl. Forward erasure correction coding and congestion control in transport. RFC 9265, 2022.

[8] Shuo-Yen Robert Li, Raymond W. Yeung, and Ning Cai. Linear network coding. *IEEE Transactions on Information Theory*, 49(2):371–381, 2003.

[9] Emmanuel Lochin, Amine Bouabdallah, and Jérôme Lacan. Tetrys: An on-the-fly network coding protocol. RFC 9407, 2023.

[10] Jeremy Loustau. aioquic: Quic and http/3 implementation in python. https://github.com/aiortc/aioquic, 2024.

[11] Florence Jessie MacWilliams and Neil James Alexander Sloane. *The Theory of Error-Correcting Codes*. North-Holland, 1977.

[12] François Michel, Alejandro Cohen, Derya Malak, Quentin De Coninck, Muriel Médard, and Olivier Bonaventure. Flec: Enhancing quic with application-tailored reliability mechanisms. *IEEE/ACM Transactions on Networking*, 31(2):606–619, 2023.

[13] François Michel, Quentin De Coninck, and Olivier Bonaventure. Quic-fec: Bringing the benefits of forward error correction to quic. In *IFIP Networking*, 2019.

[14] David A. Patterson, Garth Gibson, and Randy H. Katz. A case for redundant arrays of inexpensive disks (raid). In *ACM SIGMOD*, 1988.

[15] Costin Raiciu, Mark Handley, and Damon Wischik. Coupled congestion control for multipath transport protocols. RFC 6356, 2011.

[16] Irving S. Reed and Gustave Solomon. Polynomial codes over certain finite fields. *Journal of the Society for Industrial and Applied Mathematics*, 8(2):300–304, 1960.

[17] Vicky Sharma, Shivkumar Kalyanaraman, Koushik Kar, K. K. Ramakrishnan, and Vijaynarayanan Subramanian. Mplot: A transport protocol exploiting multipath diversity using erasure codes. In *IEEE INFOCOM*, pages 121–125, 2008.

[18] Yang Song, Yaqi Yuan, and Yubin Chen. Research on mptcp protocol based on erasure codes. In *International Conference on Intelligent Information Processing (IIP)*, pages 276–282, 2022.

[19] Jay Kumar Sundararajan, Devavrat Shah, Muriel Médard, Szymon Jakubczak, Michael Mitzenmacher, and João Barros. Network coding meets tcp: theory and implementation. *Proceedings of the IEEE*, 99(3):490–512, 2011.

[20] Shouxi Tseng, Yuchao Zhang, Bingyang Liu, Guo Chen, and Yifei Yuan. Codedbulk: Inter-datacenter bulk transfers using network coding. In *NSDI*, 2021.

## A Sender and Receiver Architecture

**Sender.** The application writes data to a standard QUIC connection (aioquic). Our coding layer intercepts outgoing QUIC datagrams before they reach the UDP socket. Datagrams are buffered until $k$ have accumulated, forming one coding group. The $k$ datagrams are

concatenated (with per-datagram length prefixes) and encoded into $n$ shares using a systematic Reed-Solomon code over $\mathrm{GF}(2^8)$ with a Vandermonde generator matrix. Each share is prepended with a 12-byte header carrying the group index (4 bytes), share index (1 byte), $k$ (1 byte), $n$ (1 byte), and per-datagram lengths (2 bytes each for up to $k$ datagrams). All $n$ shares are sent to the same destination address over a single UDP socket. The sender requires no knowledge of the underlying path topology; in our ns3 evaluation, ECMP at the first-hop gateway distributes shares across parallel links.

**Receiver.** Incoming coded packets are demultiplexed by group index into per-group buffers. As shares arrive, the receiver tracks which share indices have been received. Once $k$ distinct shares for a group are present, the receiver selects the corresponding $k$ rows of the Vandermonde matrix, inverts the resulting $k \times k$ submatrix over $\mathrm{GF}(2^8)$ via Gauss-Jordan elimination, and multiplies to recover the original concatenated datagrams. The per-datagram length fields from the header split the decoded payload back into individual QUIC datagrams, which are injected into aioquic's receive path. Shares arriving after the first $k$ are discarded. Groups that never accumulate $k$ shares are eventually treated as lost by QUIC's loss detector, triggering standard retransmission.

**CodingAwareReno.** The coding-aware congestion controller is a subclass of aioquic's Reno implementation. It overrides the byte-accounting methods to inflate all values by $n/k$. When a packet of $b$ application bytes is sent, the controller records $b \cdot n/k$ wire bytes in flight. Acknowledgments and losses decrement bytes in flight by the same inflated amount. The AIMD parameters are unchanged from standard Reno (multiplicative decrease factor 0.5, additive increase of 1 MSS per RTT, initial window of $10 \times$ MSS). The effect is that the congestion window limits total wire throughput to what a standard Reno flow would produce, satisfying RFC 9265 [7].