Tokens, Not Packets: Rethinking the Multipath QUIC Scheduling Interface

Hendrik Cech
Technical University of Munich
Germany

Patrick Bokelmann
Technical University of Munich
Germany

Nitinder Mohan
Delft University of Technology
Netherlands

Abstract

The Multipath QUIC extension has enabled a growing class of stream-aware schedulers that exploit QUIC's multiplexed streams, mixed reliable and unreliable delivery, and explicit control frames. Progress on these ideas is bottlenecked by an interface inherited from Multipath TCP, in which the scheduler is a function called during packet generation to pick a path. This coupling yields invasive, library-specific scheduler implementations, hinders application-steered scheduling, and makes fair cross-scheduler comparison impractical. We present an up-front, token-based MPQUIC scheduling interface designed primarily for research use, guided by three design choices: unified up-front decisions that address path selection, stream assignment, and duplication jointly; inspectability of scheduling policy as persistent data structures; and mechanism safety through a boundary that preserves protocol invariants. The scheduler reads transport state through a mediating *ConnState* object and writes policy into per-path, multi-level queues of abstract tokens that the library drains into packets. We prototype the interface in Cloudflare's quiche, reproduce published scheduler behavior, and demonstrate a new degree of freedom—explicit control frame routing—that reduces median stream completion time by 138 ms in our evaluation.

CCS Concepts

• **Networks** → **Transport protocols**; Network experimentation.

Keywords

MPQUIC, Multipath, QUIC, Scheduling, Scheduler

ACM Reference Format:
Hendrik Cech, Patrick Bokelmann, and Nitinder Mohan. 2026. Tokens, Not Packets: Rethinking the Multipath QUIC Scheduling Interface. In *ACM/IRTF Applied Networking Research Workshop (ANRW '26), July 20, 2026, Vienna, Austria.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3822163.3827925

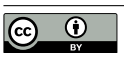
1 Introduction

Multipath QUIC (MPQUIC) [18] is on track for standardization as QUIC's [14] multipath extension, bringing simultaneous use of multiple network paths to the growing QUIC ecosystem. Unlike Multipath TCP [7], whose single ordered byte stream makes scheduling a question of *which path carries the next packet*, MPQUIC inherits QUIC's richer frame model: multiplexed `STREAM`s with priorities [22], unreliable `DATAGRAM`s [23], and explicit control frames that must themselves be scheduled. A growing body of *stream-aware* MPQUIC schedulers [11, 20, 24, 25, 27] derives decisions from frame and stream priority, not just from path state, and draws on cross-layer signals such as device-level link state.

In practice, progress on these ideas is bottlenecked by how today's MPQUIC libraries expose scheduling. The scheduler is injected into the packet-generation pipeline and consulted once per outgoing packet to pick a path; the interface inherited from MPTCP is insufficient to fully explore MPQUIC's richer decision space. Three consequences follow. First, integrating a new scheduler requires invasive, library-specific modifications that must accommodate each library's internal structure and state representation; some libraries additionally make scheduling sub-decisions, such as which stream to draw from next, before the scheduler is even consulted, constraining which policies can be expressed. Second, injecting application-provided signals typically requires further invasive changes, because the interface offers no access to the scheduler. Third, because implementations do not port across libraries, fair cross-scheduler comparison on a common substrate is largely absent from the literature.

We design an MPQUIC scheduling interface primarily for research use, guided by three design choices. *Unified, up-front decisions:* a single scheduler invocation addresses path selection, stream assignment, duplication, and reinjection jointly, rather than splitting them across layered per-packet callbacks. *Inspectability:* scheduling decisions are expressed as persistent data structures rather than as control flow during packet generation, making the intended policy visible as a single object. *Mechanism safety:* the data-structure boundary gives the scheduler full control over sending decisions while letting the library enforce protocol invariants without exposing its mutable state. The scheduler is an ordinary program with unconstrained computation, not a domain-specific script [8]. Guided by these choices, we present the following contributions:

(1) A scheduling interface for MPQUIC that decouples scheduling *policy* from transport *mechanism* (§3). On each invocation, the scheduler reads a mediating *ConnState* object (a view of path, stream, datagram, and control frame state), and expresses its policy by populating two per-path structures: a control frame queue and a multi-level token queue holding abstract `Stream` and `Datagram` tokens. The library drains these structures in priority-level order to assemble packets, transparently enforcing protocol constraints. Unlike per-packet interfaces, a single scheduler invocation specifies policy for many subsequent packets: the token queues persist across sending rounds and are edited only when the policy's inputs change.
(2) A survey of the MPQUIC/MPTCP scheduler design space (§2) and a series of case studies (§4) showing that our interface expresses per-packet, up-front, redundancy-based, and cross-layer schedulers under one abstraction.



This work is licensed under a Creative Commons Attribution 4.0 International License.
ANRW '26, Vienna, Austria
© 2026 Copyright held by the owner/author(s).
https://doi.org/10.1145/3822163.3827925

(3) A prototype of the interface in Cloudflare's quiche [2, 4] that reproduces published scheduler behavior and reveals a new degree of freedom — explicit control frame routing — that reduces median stream completion time by 138 ms in our testbed (§5). The implementation is available as open-source [10].

2 Scheduling in Multipath QUIC

Multipath QUIC (MPQUIC) lifts the single-bytestream assumption that defined a decade of multipath scheduling research. We first survey the resulting design space of MPQUIC schedulers and then argue that current implementations cannot accommodate it: integrating a new scheduler requires invasive, per-library modifications, no framework enables fair cross-scheduler comparison, and applications have no first-class way to influence scheduling decisions.

Multipath Transport in QUIC. QUIC [14] multiplexes concurrent streams into a single connection and identifies connections by Connection IDs rather than 4-tuples, enabling migration across network changes. The Multipath QUIC extension [18] generalizes this by letting endpoints use multiple active paths simultaneously, each with its own congestion controller and packet number space. A QUIC packet carries one or more *frames*: reliable data in variable-length `STREAM` frames, unreliable data in `DATAGRAM` frames [23], and everything else in what we call *control frames*. Most control frames can be sent on any path (e.g., `ACK`s, `MAX_DATA`); `PATH_CHALLENGE` and `PATH_RESPONSE` are the notable exceptions and are bound to the path they validate. This frame model is a qualitative departure from MPTCP [7], which carries a single ordered byte stream per connection and folds control information into the TCP header. The influential body of MPTCP scheduling work [6, 13, 15, 17] is accordingly *packet-level*. MPQUIC's multiplexed streams, prioritized data, mixed reliable/unreliable delivery, and explicit control frames make the data-agnostic view insufficient: scheduling decisions now depend on frame type, stream identity, and stream priority [22], not just path state.

The MPQUIC Scheduler Design Space. The richer frame model has led to a diverse set of MPQUIC schedulers that differ along two largely orthogonal axes, which we summarize in Table 1: *when* they decide and *what* they decide over.

Along the temporal axis, schedulers are either *per-packet* or *up-front*. Per-packet schedulers select a path (and occasionally a stream) for the immediate next packet. Up-front schedulers instead derive an abstract assignment that is reused for many packets and only revisited on notable events such as a new stream or a significant path change. DAPS [15] generates a schedule spanning the least common multiple of forward delays, PStream [27] allocates data volume per stream according to priority, and MuSher [26] probes for a throughput-optimal ratio between paths.

Along the granularity axis, schedulers decide at the level of *paths*, *streams*, or *stream ranges*. Path-granularity schedulers pick a path per packet and ignore content. Stream-granularity schedulers additionally pick which stream should fill that packet. Stream-range schedulers go further and address specific byte ranges independently of QUIC's ordinary transmission logic: SR-MPQUIC [20] and EdAR [9] duplicate prioritized ranges across paths to improve reliability, while XLINK [31] and Chorus [19] re-inject inflight ranges on alternate paths before QUIC's loss detection would trigger.

Table 1: MPQUIC/MPTCP schedulers span two largely orthogonal axes (invocation mode and decision granularity) with a third emerging axis (information source). A uniform interface must accommodate all three.

Scheduler	Mode	Granularity	Info. source
ECF [17]	per-packet	path	transport
BLEST [6]	per-packet	path	transport
STTF [13]	per-packet	path	transport
Peekaboo [30]	per-packet	path	transport
QAware [28]	per-packet	path	below-transport
STORM [11]	per-packet	path	below-transport
SA-ECF [25]	per-packet	stream	app (priority)
Monty [24]	per-packet	stream	app (utility)
XLINK [31]	per-packet	stream range	app (QoE)
Chorus [19]	per-packet	stream range	app (ABR)
DAPS [15]	up-front	path	transport
MuSher [26]	up-front	path	transport
PStream [27]	up-front	stream	app (priority)
SR-MPQUIC [20]	up-front	stream range	app (priority)
EdAR [9]	up-front	stream range	transport

A third, emerging axis is *information source*. Classical schedulers use only transport-layer signals (RTT, congestion window). Stream-aware schedulers lift upper-layer, application-provided information into the transport through HTTP priorities [25, 27], redundancy-protected streams [20], and custom utility functions [24]. Others reach *below* the transport: QAware [28] reads device driver queue occupancy to detect congestion at sub-RTT granularity; STORM [11] tracks wireless SNR to promptly deactivate failing paths. A uniform scheduling interface must therefore not only expose transport state but also provide access to cross-layer information.

Architectural Limitations of Current MPQUIC Stacks. Despite this diversity, prevailing MPQUIC libraries [1, 3–5, 12, 29] are structured around a single assumption inherited from MPTCP: the scheduler is a function called during packet creation that picks a path. Schedulers are therefore tightly coupled to the library's packet-generation pipeline, so that new schedulers require invasive, library-specific modifications. Libraries differ in their state representation and perform state modifications at different stages of the sending loop, e.g., selecting which stream to draw from next before the scheduler is even consulted. Therefore, a scheduler written for one library does not translate to another without substantial re-engineering. The absence of a shared interface has made direct comparison across schedulers effectively impossible. Reported results are entangled with implementation choices of the host library, and to our knowledge no independent, cross-scheduler comparison exists in the literature. Finally, applications cannot participate in scheduling without either patching the transport library or working around it above the library. Monty [24], for example, works around the library by selectively delaying writes to the library to enforce a priority scheme. Patching the library, in turn, tightly couples the application with a specific customized dependency, hindering the migration to other QUIC implementations.

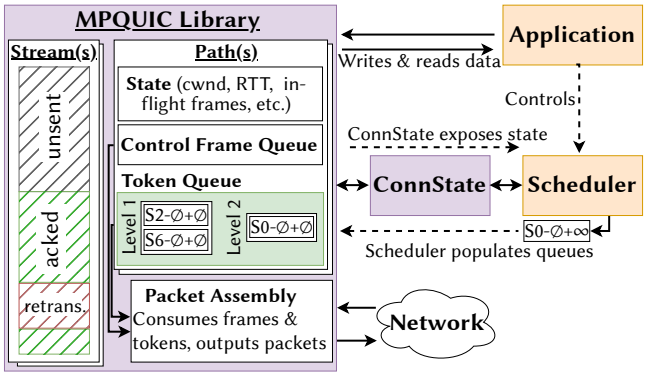


Figure 1: The *ConnState* object mediates access of the application-controlled scheduler to the QUIC state.

Related Work. The closest prior work is ProgMP [8], which enables MPTCP scheduler programming in a kernel-resident DSL with a fixed vocabulary of path properties, helper functions, and queues (send, in-flight, reinjection). We share ProgMP's goal of a centralized scheduling specification but differ on three points that are forced by the move from MPTCP to MPQUIC. ProgMP reasons over opaque packets; MPQUIC scheduling must distinguish `STREAM`, `DATAGRAM`, and control frames and address stream ranges directly. ProgMP restricts expressiveness via DSL semantics to remain safe in the kernel; we target user-space libraries and let the scheduler be an ordinary program. ProgMP channels the access of applications to schedulers through integer registers and custom packet properties; we build on the full expressiveness available from a user-space scheduler. Other notable efforts include the early-stage eBPF-based MPTCP schedulers in the Linux kernel [21], which are constrained by the eBPF runtime and operate on MPTCP.

3 A Token-Based Scheduling Interface

Our interface separates scheduling *policy* from transport *mechanism*. The scheduler is an application-provided object that, on each invocation, expresses its policy by populating two data structures per path: a *control frame queue* and a *multi-level token queue*. The MPQUIC library then drains these structures to produce packets, handling all protocol-level concerns (flow control, congestion control, frame packing, encryption) transparently. The scheduler never constructs frames or directly touches QUIC state; it reads connection state through a mediating *ConnState* object and writes decisions as abstract *tokens* that the library translates into `STREAM` and `DATAGRAM` frames. Figure 1 shows the resulting architecture and Table 2 the token vocabulary; we walk through both below.

Invocation. QUIC implementations run a state machine where state changes are triggered by incoming packets, application writes, and timeouts; each trigger may require sending new frames. Our scheduler is invoked after such a state change but before the library generates the next batch of outgoing packets. Concretely, it is called on events that may change what should be sent, including congestion window updates (from ACKs or losses), new application writes, and PTOs. Scheduling is only performed for Application-epoch packets; handshake packets are produced by the library's native path. Our presentation assumes a single-threaded library in which triggers, scheduling, packet assembly, and transmission are sequential, but the interface does not inherently rely on this.

Table 2: The six modes of a `Stream` token with a fixed `stream_id`. Symbol ∅ denotes the empty value.

`offset`	`length`	Behavior
∅	`None`	Emit one `STREAM` frame from the lowest queued offset; drop.
∅	N	Emit up to N bytes from the lowest queued offset; requeue with decremented `length`, drop at 0.
∅	∞	Emit from the lowest queued offset; always requeue until stream termination (used by MinRTT).
O	`None`	Emit one `STREAM` frame starting at O; drop.
O	N	Emit frames covering $[O, O{+}N)$; if bytes at O have already been ACKed, `offset` advances to the smallest buffered offset below $O{+}N$. Requeue with updated `offset`/`length`, drop on $N = 0$.
O	∞	Traverse the stream from O; `offset` advances past ACKed bytes; persists until stream terminates.

Table 3: The behavior of the `Datagram` token depending on its `count` parameter.

`count`	Behavior
N	Emit up to N datagrams from the send queue; requeue with decremented `count`, drop at 0.
∞	Emit a datagram if the send queue is non-empty; always requeue until connection termination.

ConnState. *ConnState* is the scheduler's sole view of the connection. It exposes per-path state (RTT, congestion-window occupancy, in-flight frames, PTO status), per-stream state (open sending streams, byte ranges queued for sending, acked ranges, flow-control limits), and per-connection datagram state (number and size of queued datagrams). It additionally surfaces the list of control frames that the library has determined must be sent in this round; the scheduler must assign all of them and must not introduce others, a property the library can verify. The path-specific `PATH_CHALLENGE` and `PATH_RESPONSE` frames are assigned automatically to the corresponding paths. *ConnState* is the only object the scheduler mutates: it exposes the per-path control frame queues and multi-level token queues as write targets, along with helper functions that, e.g., estimate the congestion-window cost of a queued `Stream` token (accounting for QUIC and crypto overhead). This allows per-packet schedulers to simulate sending decisions without replicating the library's packet-accounting logic.

Token Queues and Levels. Each path carries two structures. The *control frame queue* holds the path-assignable control frames the scheduler has routed to that path; a frame may be duplicated across paths, since duplicates are silently dropped by the peer as part of ordinary QUIC deduplication. The *multi-level token queues* hold `Stream` and `Datagram` tokens keyed by an integer *level*. Levels define priority classes that impose ordering both within and across paths: when the library produces packets, it repeatedly selects the

lowest level that contains an available token on any path and processes that token. Ties (the same lowest level available on multiple paths) are broken by path identifier. Within a level, processing is round-robin: a token at the head of the queue is either consumed and dropped or, depending on its `length` (Table 2), requeued at the tail. Each queued token carries an *availability* flag. A token is marked *unavailable* when it cannot presently produce a frame, e.g., if all queued data has been sent, or if the path's congestion window is exhausted, and is skipped until the condition clears. When every token in a level is unavailable, the library advances to the next higher level.

Control frame queues are cleared before every scheduler invocation because their contents (ACK ranges, flow control updates) go stale across rounds; the scheduler reassigns them each time. Token queues, in contrast, *persist* across invocations. This is the central design choice that makes the interface up-front: a scheduler can install a policy once and have it honored over many send rounds, revisiting it only when conditions change (new streams, RTT reordering, etc.).

Data Tokens. A `Stream` token carries a mandatory `stream_id` and two optional fields, `offset` and `length`, whose combinations yield six transmission modes (Table 2). The two fields are orthogonal. Field `offset` chooses *where* emission begins: ∅ delegates to the library's send queue (picking the lowest queued offset, either a retransmission or unsent data), while a concrete value O addresses an explicit byte offset. Field `length` controls *how much* data is emitted and the token's lifetime (∅: dropped after use; N: bounded; ∞: persistent until the stream terminates). Persistent tokens are the idiom for a standing policy; explicit-offset tokens enable duplication and targeted reinjection.

A few points are not reflected in the table. A `Stream` token is dropped when its stream is closed (finished or reset), regardless of mode; persistent tokens for that stream are not requeued. For explicit-offset tokens, the library may find that the data at `offset` has already been acknowledged and discarded. In that case, `offset` is advanced to the smallest buffered offset below the token's upper bound `offset + length`. If the range is entirely acknowledged, the token is dropped. The requeued token reflects what actually remained to send. For example, consider a token with `offset` 1000 and `length` 3000. If bytes 0–2000 of its stream are already acknowledged, and only 500 B of the remaining range fit into the current packet, the token is requeued with `offset` 2500 and `length` 1500. When `offset` is ∅ and a retransmission is pending, the library sends exactly the queued range needed to cover the loss; data beyond that range is no longer buffered and is not re-read.

A `Datagram` token is specified with a `count` parameter that controls how many `DATAGRAM` frames it will emit. The token is flagged as unavailable if the datagram send queue is currently empty. The token is requeued with a decremented `count` until it reaches 0.

Packet Generation. Once the scheduling function returns, the library iteratively builds packets by selecting the path whose token queue holds the lowest available level. It first emits that path's pending control frames, then draws tokens from the level queue and converts them into `STREAM` and `DATAGRAM` frames until the packet is full. Each token is then dropped or requeued per Tables 2 and 3, potentially with its availability flag cleared. Protocol constraints are enforced during this conversion rather than being exposed to the scheduler. If flow control prevents emitting stream data, the offending `Stream` token is requeued as unavailable. If a path's congestion window becomes exhausted, every remaining token on that path is marked unavailable in bulk, and packet generation moves to the next eligible path.

Portability Requirements. The interface places two requirements regarding stream data management on a host library: (1) a per-stream accounting of byte ranges queued for sending (unsent data and loss-recovery re-queues), and (2) the ability to read from arbitrary offsets in the stream send buffer, including in-flight data that has been sent but not yet acknowledged. Both follow from the QUIC specification: loss recovery already requires tracking and re-queuing outstanding ranges, and data must be buffered until acknowledgement. How readily a library meets them depends on its stream bookkeeping. The remaining integration work is in extracting connection state into *ConnState* and wiring the token-drain loop into the packet builder. The effort likewise varies with library internals: quiche, for instance, interleaves control-frame determination with packet assembly, which we had to separate. No changes to congestion control, loss detection, or existing APIs (e.g., stream priorities) are necessary.

Design Rationale. A scheduling interface can vary in the granularity it controls, when decisions are placed relative to packet construction, and how they are expressed. We chose an up-front, data-structure-based interface at frame granularity for three reasons. First, the token queues capture the full scheduling policy as a single, inspectable object rather than scattering it across many per-packet decisions. Second, the interface unifies path selection, stream assignment, duplication, and reinjection in a single decision made before the sending process starts. Current per-packet interfaces split these concerns: the scheduler picks a path, stream selection is library-internal, and duplication requires ad-hoc intervention. Deciding up-front lets the scheduler reason over the full connection state and address all concerns jointly. Third, the data-structure boundary lets the library enforce protocol invariants without exposing mutable state to the scheduler. Persistent queues additionally amortize decision cost, and the simulation idiom (§4) shows that per-packet policies remain expressible.

4 Interface Usage

We now show how the interface is used, building up from the simplest case to progressively richer policies. We close by showing that even a per-packet scheduler, the paradigm our up-front model was *not* designed for, can be expressed via a short simulation idiom.

Stream-Aware MinRTT. The MinRTT scheduler sends on the path with the lowest RTT and free congestion window. In MPTCP, this is a per-packet path choice; in MPQUIC, a stream-aware MinRTT must additionally decide *which stream* fills each packet. We express both the scheduling policy and the stream-awareness in a single persistent token assignment, shown in Figure 2. For each sending stream, we push one `Stream` token with `offset` ∅ and `length` ∞: the token draws from the library's send queue (inheriting its retransmission behavior) and is requeued after each emission until the stream closes. The level mechanism unifies cross-path minRTT ordering and stream prioritization. Streams occupy sequential

levels on the faster path (sorted by priority, starting at level 1), while the slower path mirrors this priority structure offset above the faster path's last level. Packet generation thus drains the faster path's queue (in priority order) before touching the slower path's. Same-priority streams are processed in round-robin order (§3).

Re-Invocation and Queue Freshness. The scheduler is invoked on every significant state change. Naively rebuilding the token queues on every invocation would break round-robin fairness by pushing waiting streams back to the tail. However, because policies like minRTT use persistent tokens, they only need to rebuild queues during structural changes (new streams, paths, or priorities). Routine updates (ACK arrivals, congestion window growth) are handled seamlessly by the existing queue state. Since structural changes are infrequent, fairness is not meaningfully impacted. Schedulers that must react to routine updates can avoid rebuilds by editing existing tokens in place, though this increases complexity.

Redundant Transmission. Redundant schedulers duplicate stream bytes across paths to minimize flow-completion time under loss. Full redundancy is expressed by pushing a `Stream` token with `offset` 0 and `length` ∞ per stream on each path. Unlike MinRTT, where offset ∅ delegates to the send queue to transmit bytes once, offset 0 addresses the stream directly. When requeued, the token's offset advances past emitted or already-acknowledged bytes, ensuring each path traverses the stream sequentially and natively skips ranges the peer has received. Selective redundancy (e.g., SR-MPQUIC [20]) uses the same mechanism: push explicit-offset persistent tokens for prioritized streams on all paths, and MinRTT-style queue-driven tokens for the rest. Tokens express byte ranges rather than packets, natively accommodating different path MTUs: the library automatically fragments data into MTU-sized STREAM frames during packet generation.

Selective Reinjection. Persistent tokens express standing policy; bounded explicit-offset tokens express timely interventions. Consider a lost stream frame that originally carried data at offset O with length N. When the library detects this loss, the range is requeued for sending, and the MinRTT token's next emission will naturally carry the retransmission. A scheduler that wants the retransmission to race across all paths to minimize HoL blocking, as in XLINK [31] and Chorus [19], can push a bounded token explicitly targeting the lost `offset` O and `length` N on every path. Two conventions make this fit cleanly into the rest of the schedule. First, we reserve level 0 for interventions: the scheduler's standing policy uses levels ≥ 1, and any token at level 0 is drained before anything else on any path. Second, the scheduler does not have to track the token completion since bounded tokens drop automatically after producing stream range O to $O + N$. The same pattern implements early reinjection (send the duplicate before loss detection triggers) by pushing the bounded token at a scheduler-chosen time, without waiting for the library's loss signal.

Control Frame Scheduling. In MPTCP, control information piggybacks on every packet header; in MPQUIC, it lives in explicit frames that the scheduler must route. The interface exposes the library's pending control frames (ACKs, `MAX_DATA`, etc.) through *ConnState* and requires the scheduler to assign each frame to one or more paths' control frame queues. This enables control frame routing decisions, e.g., sending acknowledgements on the matching receive path or duplicating critical updates across paths.

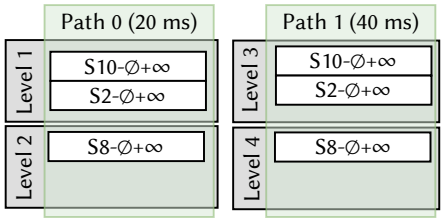


Figure 2: Persistent token policy of stream-aware MinRTT (fast path 0). Tokens are shown as `S<stream_id>-<offset>+<length>`. Changing `offset` to 0 results in a fully-redundant scheduler.

Per-Packet Scheduling via Simulation. The idioms above are natural fits for up-front scheduling. The potentially difficult case is a scheduler like SA-ECF [25], whose decision for packet $k + 1$ depends on the congestion window and stream state left by packet k. We accommodate per-packet schedulers with a short simulation idiom that reuses the up-front interface without modification.

Our SA-ECF implementation opens a scheduling round by reading each path's available congestion window from *ConnState*. It then runs a per-packet loop against a simulated copy of that state: each iteration picks the path minimizing stream completion time, and appends one `Stream` token with `offset` ∅ and `length` ∅ (to schedule a single frame) to that path's queue at a strictly increasing level, enforcing the simulation's chosen packet order. The loop decrements the simulated window by the token's projected size (including frame, QUIC, and crypto overhead) using a *ConnState* helper, and terminates when every simulated window is exhausted.

The cost of this translation is bounded: the simulation performs the same arithmetic the per-packet scheduler would have performed anyway, just clustered into a single call rather than spread across the sending loop. We quantify this in §5.

5 Implementation & Evaluation

We implement our scheduling design on top of an experimental MPQUIC fork [4] of Cloudflare's `quiche`, written in Rust. The fork predates the latest MPQUIC draft, but no intervening change affects our scheduling model. Our additions amount to ≈ 3500 LOC: roughly 2300 lines for the scheduler module (token queues, *ConnState*, scheduling logic) and 1250 lines of integration with quiche's sending pipeline (hooking state changes, feeding tokens to the packet builder, and surfacing connection state). A scheduler implemented on the interface is considerably smaller; our MinRTT scheduler is 150 LOC. We implemented the schedulers from §4 (stream-aware MinRTT, redundant, reinjection, ECF, SA-ECF) on our token-based interface, and additionally MinRTT and ECF on a conventional per-packet interface for comparison. We evaluate them in Mininet [16] with two emulated 10 Mbps paths of varying RTT, using a custom iperf-like application that transfers 100 KB over 16 equally-weighted streams. We report 30 runs per configuration.

Runtime Performance. We instrument scheduler invocations and compare per-packet and up-front implementations of MinRTT and

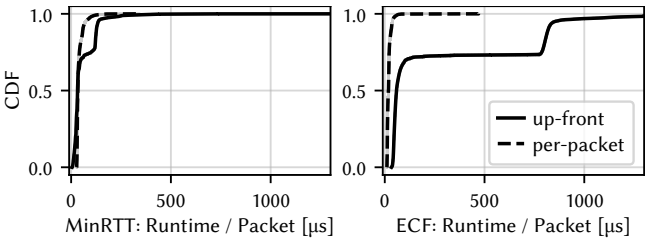
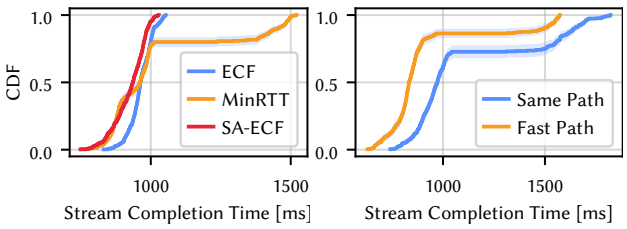


Figure 3: The runtime of per-packet and up-front implementations normalized by emitted packets.



(a) (SA)ECF. **(b) MinRTT ACK policies.**

Figure 4: Demonstration of up-front (a) `STREAM` and (b) control frame schedulers.

ECF in Figure 3. Median execution times are 126/800 µs for up-front MinRTT/ECF against 32/15 µs for their per-packet counterparts. The per-packet scheduler runs many times per send round while the up-front scheduler runs once; the figure therefore normalizes by packet count. The two MinRTT variants have comparable per-packet runtime (median 33 µs each), because the up-front MinRTT performs minimal work per invocation: it checks for structural changes and edits the token assignment only when paths reorder. The ECF variants diverge more sharply: the up-front median is only 40 µs higher, but its upper quartile reaches 786 µs and beyond, reflecting the cost of the SA-ECF-style simulation idiom on long send rounds. The runtime of the ECF variants diverges noticeably: per-packet ECF is cheap (15 µs) because it evaluates a single path-selection formula per call, while up-front ECF runs the SA-ECF simulation loop that iterates over the entire congestion window, accumulating cost proportional to the send round length. This highlights that up-front MinRTT is a natural fit for the token model (install once, reuse), while up-front ECF must simulate per-packet decisions, paying the cost of the simulation idiom described in §4. The overhead of *ConnState* indirection and up-front token assignment is real but bounded, and amortizes across packets within a round. We have not optimized the implementation and expect meaningful gains from caching *ConnState* views and specializing the token-push path.

Scheduler Comparison. Figure 4a shows the stream completion times (SCTs) of MinRTT, ECF, and SA-ECF with the RTTs of the two paths set to 10 and 100 ms, respectively. ECF and SA-ECF share their core path-selection algorithm, but SA-ECF's stream-awareness improves its SCTs by a median of 27 ms. MinRTT performs worst, particularly because of tail SCTs that are up to 500 ms higher compared to the other two schedulers. While the path RTTs were static, queuing delay changed the RTT order of the paths, which forced rebuilding the token queue.

Figure 4b isolates control frame scheduling with the RTTs of the two paths set to 10 and 25 ms, respectively. We keep the underlying MinRTT scheduler fixed and vary only the ACK assignment policy: *Same Path* returns each ACK on the path that received the acknowledged data, while *Fast Path* sends all ACKs on the lowest-RTT path. The latter reduces median SCT by 138 ms, exposing a scheduling axis that existing MPQUIC interfaces do not. We apply this scheduling policy to both endpoints. Although the receiver sends no stream data, it maintains path RTT estimates via occasional `PING` frames that quiche bundles with its ACKs (following RFC 9000 [14] to limit unacknowledged packets).

6 Conclusion & Future Work

We presented a token-based scheduling interface for Multipath QUIC that decouples policy from packet generation, expresses schedulers from across the design space, and treats control-frame routing as a first-class concern. Our quiche-based prototype [10] reproduces published scheduler behavior and demonstrates improvements through control-frame routing, a degree of freedom no existing interface exposes.

Scope. The interface targets scheduling decisions at the frame level above the packet boundary through the token abstraction. It does not grant control over packetization, which remains with the library. This means, for example, that requiring fixed-size packets for traffic analysis resistance would need a library-level PADDING policy, which needs to be configured orthogonally to the scheduling interface. Additionally, schedulers that must react on sub-RTT timescales, for example to per-packet pacing events, need triggers beyond our current set of ACKs, new writes, and PTOs.

Future Work. We are exploring WebAssembly-based modularization so that endpoints can securely exchange custom schedulers over an active connection without per-library modifications. Because scheduling decisions are data in the token queues rather than control flow, they can be asserted on, diffed, and replayed, enabling scheduler debugging and compliance testing. We are porting the interface to other QUIC substrates and developing a compliance test suite against a reference token-drain specification.

Acknowledgements

This work was funded in part by the German Federal Ministry for Digital and Transport (BMDV) within the InnoNT program under grant 19OI22017O and National Growth Fund through the Dutch 6G flagship project "Future Network Services".

Artifacts

Our implementation and supporting material is openly available at https://github.com/hendrikcech/anrw26-tokens-not-packets [10].

References

[1] Alibaba. 2026. *xquic*. Retrieved 2026-04-17 from https://github.com/alibaba/xquic

[2] Cloudflare. 2026. *quiche*. Retrieved 2026-04-17 from https://github.com/cloudflare/quiche

[3] Quentin De Coninck. 2017. *qdeconinck/mp-quic*. Retrieved 2026-04-17 from https://github.com/qdeconinck/mp-quic

[4] Quentin De Coninck. 2026. *cloudflare/quiche: Multipath support with non-zero length Connection IDs*. Retrieved 2026-04-17 from https://github.com/cloudflare/quiche/pull/1310

[5] EricssonResearch. 2026. *aioquic-multipath*. Retrieved 2026-04-17 from https://github.com/EricssonResearch/aioquic-multipath

[6] Simone Ferlin, Özgü Alay, Olivier Mehani, and Roksana Boreli. 2016. BLEST: Blocking estimation-based MPTCP scheduler for heterogeneous networks. In *2016 IFIP Networking Conference (IFIP Networking) and Workshops*. IEEE, 431–439.

[7] Alan Ford, Costin Raiciu, Mark J. Handley, Olivier Bonaventure, and Christoph Paasch. 2020. TCP Extensions for Multipath Operation with Multiple Addresses. RFC 8684. doi:10.17487/RFC8684

[8] Alexander Frömmgen, Amr Rizk, Tobias Erbshäußer, Mira Weller, Boris Koldehofe, Alejandro Buchmann, and Ralf Steinmetz. 2017. A programming model for application-defined multipath TCP scheduling. In *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference* (Las Vegas Nevada). ACM, 134–146. doi:10.1145/3135974.3135979

[9] Jiangping Han, Kaiping Xue, Jian Li, Rui Zhuang, Ruidong Li, Ruozhou Yu, Guoliang Xue, and Qibin Sun. 2023. EdAR: An Experience-Driven Multipath Scheduler for Seamless Handoff in Mobile Networks. 22, 10 (10 2023), 6839–6852. doi:10.1109/TWC.2023.3246082

[10] Hendrik Cech. 2026. *Tokens, Not Packets: Rethinking the Multipath QUIC Scheduling Interface*. Retrieved 2026-06-22 from https://github.com/hendrikcech/anrw26-tokens-not-packets

[11] Liekun Hu and Changlong Li. 2025. STORM: a Multipath QUIC Scheduler for Quick Streaming Media Transport under Unstable Mobile Networks. (2025), 851–866.

[12] Christian Huitema. 2026. *picoquic*. Retrieved 2026-04-17 from https://github.com/private-octopus/picoquic

[13] Per Hurtig, Karl Johan Grinnemo, Anna Brunstrom, Simone Ferlin, Özgü Alay, and Nicolas Kuhn. 2019. Low-Latency Scheduling in MPTCP. 27, 1 (2019), 302–315. doi:10.1109/TNET.2018.2884791

[14] Jana Iyengar and Martin Thomson. 2021. QUIC: A UDP-Based Multiplexed and Secure Transport. RFC 9000. doi:10.17487/RFC9000

[15] Nicolas Kuhn, Emmanuel Lochin, Ahlem Mifdaoui, Golam Sarwar, Olivier Mehani, and Roksana Boreli. 2014. DAPS: Intelligent delay-aware packet scheduling for multipath transport. In *2014 IEEE International Conference on Communications (ICC)* (Sydney, NSW). IEEE, 1222–1227. doi:10.1109/ICC.2014.6883488

[16] Bob Lantz, Brandon Heller, and Nick McKeown. 2010. A network in a laptop: rapid prototyping for software-defined networks. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks* (New York, NY, USA) *(Hotnets-IX)*. Association for Computing Machinery, 1–6. doi:10.1145/1868447.1868466

[17] Yeon-sup Lim, Erich M Nahum, Don Towsley, and Richard J Gibbens. 2017. ECF: An MPTCP path scheduler to manage heterogeneous paths. In *Proceedings of the 13th International Conference on emerging Networking EXperiments and Technologies*. 147–159.

[18] Yanmei Liu, Yunfei Ma, Quentin De Coninck, Olivier Bonaventure, Christian Huitema, and Mirja Kühlewind. 2026. *Managing multiple paths for a QUIC connection*. Internet-Draft draft-ietf-quic-multipath-21. Internet Engineering Task Force. https://datatracker.ietf.org/doc/draft-ietf-quic-multipath/21/ Work in Progress.

[19] Gerui Lv, Qinghua Wu, Yanmei Liu, Zhenyu Li, Qingyue Tan, Furong Yang, Wentao Chen, Yunfei Ma, Hongyu Guo, Ying Chen, and Gaogang Xie. 2024. Chorus: Coordinating Mobile Multipath Scheduling and Adaptive Video Streaming. In *Proceedings of the 30th Annual International Conference on Mobile Computing and Networking* (New York, NY, USA) *(ACM MobiCom '24)*. Association for Computing Machinery, 246–262. doi:10.1145/3636534.3649359

[20] Rasmus S. Mogensen, Christian Markmoller, Tatiana K. Madsen, Troels Kolding, Guillermo Pocovi, and Mads Lauridsen. 2019. Selective Redundant MP-QUIC for 5G Mission Critical Wireless Applications. In *2019 IEEE 89th Vehicular Technology Conference (VTC2019-Spring)*. 1–5. doi:10.1109/VTCSpring.2019.8746482

[21] multipath tcp/mptcp_net_next. 2020. *BPF: packet scheduler*. Retrieved 2026-04-17 from https://github.com/multipath-tcp/mptcp_net-next/issues/75

[22] Kazuho Oku and Lucas Pardue. 2022. Extensible Prioritization Scheme for HTTP. RFC 9218. doi:10.17487/RFC9218

[23] Tommy Pauly, Eric Kinnear, and David Schinazi. 2022. An Unreliable Datagram Extension to QUIC. RFC 9221. doi:10.17487/RFC9221

[24] Alexander Rabitsch, Per Hurtig, Stefan Alfredsson, and Anna Brunstrom. 2024. Monty: A Framework for Latency-aware Multi-flow ATSSS Scheduling. In *2024 IEEE 49th Conference on Local Computer Networks (LCN)*. 1–9. doi:10.1109/LCN60385.2024.10639681 ISSN: 2832-1421.

[25] Alexander Rabitsch, Per Hurtig, and Anna Brunstrom. 2018. A Stream-Aware Multipath QUIC Scheduler for Heterogeneous Paths. In *Proceedings of the Workshop on the Evolution, Performance, and Interoperability of QUIC* (Heraklion Greece). ACM, 29–35. doi:10.1145/3284850.3284855

[26] Swetank Kumar Saha, Shivang Aggarwal, Rohan Pathak, Dimitrios Koutsonikolas, and Joerg Widmer. 2019. MuSher: An Agile Multipath-TCP Scheduler for Dual-Band 802.11ad/ac Wireless LANs. (2019), 1–16. http://dl.acm.org/doi/10.1145/3300061.3345435 ISBN: 9781450361699.

[27] Xiang Shi, Lin Wang, Fa Zhang, Biyu Zhou, and Zhiyong Liu. 2020. PStream: Priority-Based Stream Scheduling for Heterogeneous Paths in Multipath-QUIC. In *2020 29th International Conference on Computer Communications and Networks (ICCCN)*. 1–8. doi:10.1109/ICCCN49398.2020.9209682

[28] Tanya Shreedhar, Nitinder Mohan, Sanjit K Kaul, and Jussi Kangasharju. 2018. QAware: A cross-layer approach to MPTCP scheduling. In *2018 IFIP Networking Conference (IFIP Networking) and Workshops*. IEEE, 1–9.

[29] Tencent. 2026. *tquic*. Retrieved 2026-04-17 from https://github.com/tencent/tquic

[30] Hongjia Wu, Özgü Alay, Anna Brunstrom, Simone Ferlin, and Giuseppe Caso. 2020. Peekaboo: Learning-Based Multipath Scheduling for Dynamic Heterogeneous Environments. 38, 10 (10 2020), 2295–2310. doi:10.1109/JSAC.2020.3000365

[31] Zhilong Zheng, Yunfei Ma, Yanmei Liu, Furong Yang, Zhenyu Li, Yuanbo Zhang, Jiuhai Zhang, Wei Shi, Wentao Chen, Ding Li, Qing An, Hai Hong, Hongqiang Harry Liu, and Ming Zhang. 2021. XLINK: QoE-driven multi-path QUIC transport in large-scale video services. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference* (New York, NY, USA) *(SIGCOMM '21)*. Association for Computing Machinery, 418–432. doi:10.1145/3452296.3472893