ONEFRONTIER · VOL. 1, NO. 1 · NOVEMBER 2026 · ONE FRONTIER FOUNDATION · ORIGINAL RESEARCH

The Door to the Rented Box

The QUIC Tunnel Advantage: what a rental's one SSH door costs, what the measurements show, and where it stops helping

The One Frontier Research Desk
One Frontier Foundation

20 NOVEMBER 2026 · RESEARCH CURRENT THROUGH 20 NOVEMBER 2026 · ONEFRONTIER.ORG

ABSTRACT

A GPU rental marketplace drops a renter into a container on someone else's box and hands them one door: `ssh -L 8080:localhost:8080 <host>`. Every service the renter runs is reached through that provider's SSH tunnel. qqt keeps the door and its forwarding — the same `-L` and `-R` — and changes what carries the forwarded connection: one mutually authenticated QUIC connection instead of the provider's SSH session. On the house's relayed path, that carrier measured 177 against 504 Mbit/s of bulk goodput and 49.7 against 233.1 ms of keystroke echo at the median.

The gain is real and it is conditional, and it is a fact about the carrier rather than the reason for the door: it grows with latency and loss, and on a direct path at the same round trip it falls to about nine milliseconds. What QUIC does not change is the shared congestion window, and this volume is as careful about the second half of that sentence as the first.

KEYWORDS provider SSH tunnel · GPU rental · port forwarding · QUIC · ssh port forwarding · TCP-in-TCP · head-of-line blocking · congestion control · bandwidth-delay product · tunnel transport

CONTENTS SIXTEEN SECTIONS · 43 SOURCES

01 The question
02 TCP-in-TCP
03 Head-of-line blocking
04 What QUIC actually changes
05 Congestion control in the tunnel
06 Windows, buffers and loss
07 Setup cost
08 The published record
09 House measurement I
10 House measurement II
11 The 14 MB drop
12 Ergonomics
13 The security model
14 Where it does not help
15 The decision guide
16 Reproduction and bibliography
– References

01 · THE QUESTION

The question: what door a rental gives you

The reference is a GPU rental marketplace. It drops the renter into a docker container on someone else's box and hands them one door: `ssh -L 8080:localhost:8080 <host>`*. Nothing else is reachable; the provider's SSH tunnel is the route in, and the service behind it is what the rental was bought for. A file copy, a shell, a JSON-RPC call — each crosses that door as a forwarded TCP connection on one path.*

qqt answers with the same door and syntax: `-L` and `-R` bind a local port and forward it as `ssh` does. The question is what that door costs on the paths the house runs, and what changes when the forwarding is carried by a different transport.

Two ways to reach the same box

`ssh -L` binds a local port and forwards every connection accepted there to a target the far side dials. The bytes of each forwarded connection travel inside the SSH session's own TCP connection to the server, then out to the target. That is TCP carried inside TCP, and the vocabulary of this volume turns on it: the inner connection is the tunnelled flow, the outer connection is the SSH session's TCP. The relayed path adds a provider relay or jump host between client and box. The tunnel path is qqt client to qqt server, QUIC over UDP.

qqt keeps the semantics and replaces the carrier. The client opens one QUIC connection (RFC 9000) over UDP, with TLS 1.3 and mutual certificate authentication, and the certificates are the access control (`docs/architecture_qqt-protocol.md`). Each accepted TCP connection opens one bidirectional QUIC stream whose first bytes name the target host and port. Whoever receives a data stream dials that target from its own network position, so `-L` and `-R` are the same frame with the roles mirrored. FIN on one side finishes the QUIC send stream and shuts down the other side's write direction; an aborted pump resets the stream rather than ending it cleanly, so a peer never reads a failure as end-of-file.

The shape is a port forward an engineer already knows; what differs sits underneath it.

Why a stream inside a stream is the problem

Three properties of TCP compose badly when one TCP connection carries another.

Two controllers, one path. The outer connection treats every inner flow and every inner retransmission as one flow, with one congestion window, one RTT estimate and one retransmission timeout. The inner has its own. "Two TCP congestion controls … operate simultaneously and interfere each other" (Honda et al. 2005). Neither can see the other's state.

The timer ladder. A loss does not only delay bytes; the outer's retransmission timer doubles on each expiry: "The host MUST set RTO <- RTO * 2" (RFC 6298 §5.5), with a one-second floor (§2.4) and a permitted cap of at least sixty seconds (§2.5). The inner's RTT samples are taken across the outer's queue, so when the outer backs off, the inner's measured RTT rises and its own timeout rises with it. Titz's account is the canonical one: the upper layer "will queue up more retransmissions faster than the lower layer can process them", an "internal meltdown effect" (Titz 2001).

One head, every flow behind it. TCP is "a reliable, in-order, byte-stream service" (RFC 9293 §2.2). One ordered stream has one head, so a lost outer segment blocks every byte behind it, including bytes belonging to unrelated inner flows. A tunnel carrying a terminal, a bulk copy and a request-response call at once makes those three share one queue, one buffer and one backoff.

QUIC removes the second reliability layer and the cross-stream head-of-line block. Streams are independent ordered byte sequences with no ordering guarantee between them, each with its own flow control (RFC 9000 §2, §4.1). It does not remove everything. One connection still has one congestion controller and one loss-recovery context (RFC 9002 §7), and QUIC's probe timeout also doubles (RFC 9002 §6.2.1). The honest claim is narrower than "QUIC is faster": a forwarded connection no longer waits behind an unrelated flow and no longer pays a second, blind congestion controller.

What the volume measures

The volume reports the house's own benchmark record against the published literature. The record holds three kinds of measurement: bulk goodput on a relayed path, keystroke latency against the same shell reached two ways, and a direct-path pair where both transports see the same round trip.

QUANTITY	VALUE	NOTES
`bulk ceiling, TCP`	177 Mbit/s	90.4 ms relayed RTT
`bulk ceiling, qqt`	504 Mbit/s	41.5 ms direct RTT
`keystroke echo p50, ssh`	233.1 ms	120 samples per cell
`keystroke echo p50, qqt`	49.7 ms	120 samples per cell

The keystroke pair is the cleanest demonstration and the fairest to attack. Both panes ran the same `sshd` on the same box, driven by one keystroke stream, recorded at 60 frames per second with the keypress visible in frame. The shell, the key and the far end cancel. What remains is the way there: the relay is a store-and-forward hop, so a keystroke pays roughly four hops instead of two. The tunnel goes straight to the box's UDP port at 40.7 ms.

The controlled pair keeps the volume honest. On a rented EC2 instance reached directly, the path is a plain 164 ms round trip and the tunnel's is the same 164 ms, so the relay is out of the comparison and only the transport differs. There the gap is 170.8 against 179.9 ms on echo and 212.7 against 237.8 ms on a typed burst. Nine and twenty-five milliseconds. Real, repeatable, nothing like five to one.

The thesis

On the long, lossy, relayed paths the house actually runs, a TCP tunnel pays a second reliability layer and a shared head, a QUIC tunnel removes both, and the size of the resulting gain is a property of the path rather than of the protocol. The door is the same on either carrier; only what crossing it costs changes.

The three numbers that carry it are 177 against 504 Mbit/s, 49.7 against 233.1 ms, and the nine-millisecond echo advantage when the relay is gone. The first two are the finding; the third is the limit on it.

The literature agrees on the shape. Wolsing et al. (2019), measuring against TCP tuned to QUIC's own defaults, found a Speed Index difference of 87 ms on a clean 24 ms link and 8.36 s on a 760 ms link with 6 percent loss: two orders of magnitude between two paths. Google's A/B test saw Search latency fall 8.0 percent on desktop and rise 0.6 percent at the first percentile on mobile (Langley et al. 2017). A transport that helps most where the path hurts most looks like a miracle on one link and like noise on another.

What to take away

Three things. First, the mechanism is specified behaviour, quoted from the RFCs, and it costs what they say it costs. Second, the house's numbers are large and conditional, and both adjectives are load-bearing; a reader reproducing them should expect the same shape and their own two medians. Third, the decision an engineer faces is not QUIC against TCP in the abstract. It is whether the path in front of them has the latency, loss and relay hops that make the second reliability layer expensive. The chapters that follow supply the mechanism, the literature and the limits to answer that for their own path.

---

02 · TCP-IN-TCP

TCP-in-TCP: the meltdown, and the arithmetic behind it

One path, two clocks

A TCP tunnel — `ssh -L`, PPP over SSH, VTun, OpenVPN in `tcp` mode — carries a byte stream. When the payload of that stream is itself TCP, two congestion controllers govern the same path and neither can see the other. The outer treats the whole tunnel as one flow: one congestion window, one RTT estimate, one retransmission timer. The inner connection carries its own (Honda et al. 2005).

The timers are the sharp part. RFC 6298 §5.5 requires the sender to double its RTO on each expiry, and §2.4 floors it at one second, §2.5 caps it no lower than sixty. Both layers obey this. Titz described the consequence in 2001: "the upper layer will queue up more retransmissions faster than the lower layer can process them" (Titz 2001).

The two ladders compose rather than merely coexist, and the coupling is normative. An inner connection's RTT samples are taken across the outer connection's queue. The instant the outer backs off, the inner's measured RTT rises, and its RTO — `SRTT + 4 × RTTVAR` (RFC 6298 §2) — rises with it. The inner is not backing off from the path's latency. It is backing off from a number the outer's backoff manufactured. Recovery then costs the outer's full backoff interval, plus re-ramping the inner window on the inflated estimate.

The two halvings

Take the measured path: a 90.4 ms relayed round trip and a 177 Mbit/s bulk ceiling. The bandwidth-delay product is the number of bytes that must be in flight for the path to be full, and throughput is that window divided by the round trip:

$$B = \frac{W}{R_{\mathrm{tt}}} \tag{1}$$

$$W_{\mathrm{bdp}} = B \times R_{\mathrm{tt}} = 177\ \mathrm{Mbit/s} \times 0.0904\ \mathrm{s} = 16.0\ \mathrm{Mbit} = 2.0\ \mathrm{MB} \tag{2}$$

Since throughput is window ÷ RTT, 2.0 MB ÷ 90.4 ms gives back 177 Mbit/s: the window is full and the path is at its ceiling.

One loss event is observed by both layers, so it buys two cuts. Each cut scales the shared window by the controller's own decrease factor, and the ceiling falls with it:

$$W' = \beta W_{\mathrm{bdp}}, \qquad B' = \frac{\beta W_{\mathrm{bdp}}}{R_{\mathrm{tt}}} = \beta B \tag{3}$$

TABLE 1. One loss, two controllers. Each row is the shared window after the cut and the ceiling it implies at 90.4 ms. The decrease factor is the controller's own: 0.5 for Reno, 0.7 for CUBIC.

AFTER ONE LOSS	SHARED WINDOW	CEILING AT 90.4 MS
plateau	2.0 MB	177 Mbit/s
CUBIC decrease, ×0.7	1.4 MB	124 Mbit/s
Reno decrease, ×0.5	1.0 MB	88.5 Mbit/s
RTO, one 1,460 B segment	1.5 KB	129 kbit/s

The cut sizes are version-specific, not a single law: Reno halves ($\beta = 0.5$), CUBIC cuts to 0.7 ($\beta = 0.7$, RFC 9438 §3.4), and a timeout drops the window to one full-sized segment "regardless of the value of IW" (RFC 5681 §3.1). The last row is where a loss stops being expensive and becomes an outage in miniature: at 90.4 ms, a one-segment window is 1,370 times below the plateau.

The two halvings do not multiply into a quarter of the rate. They serialize in time. The outer's cut takes the path out of service for a timer interval; when it returns, the inner re-opens against an RTT the backoff inflated, so each round trip buys less than the last one did. From one segment, slow start doubles the window per round trip, so returning to the plateau costs a base-two logarithm of the window ratio:

$$n_{\mathrm{rt}} = \log_2\!\left(\frac{W_{\mathrm{bdp}}}{W_{\mathrm{seg}}}\right) = \log_2\!\left(\frac{2.0\ \mathrm{MB}}{1460\ \mathrm{B}}\right) \approx 10.5 \quad \Rightarrow \quad 10.5 \times 90.4\ \mathrm{ms} \approx 0.94\ \mathrm{s} \tag{4}$$

That assumes no second loss.

The floor is what makes the ladder bite. One second of RTO is eleven relayed round trips of silence, or twenty-four on the 41.5 ms direct path. A nested controller pays its loss-recovery amplification in proportion to the ratio of the two retransmission timers: the inner fires whenever its own timer expires, and the outer keeps re-sending what the inner already has in flight, so the copies written into the outer socket per outer expiry multiply:

$$A_{\mathrm{rto}} = \frac{RTO_{\mathrm{outer}}}{RTO_{\mathrm{inner}}} = \frac{8\ \mathrm{s}}{1\ \mathrm{s}} = 8 \tag{5}$$

An outer sitting at 8 s against an inner at 1 s therefore lets the inner fire eight times per outer expiry: eight retransmission copies into a socket that cannot drain, each one resetting the inner window to a single segment. The outer already guarantees delivery, so every one of those copies is redundant traffic competing with the data it claims to recover.

The queue at the head

TCP is "a reliable, in-order, byte-stream service" (RFC 9293 §2.2). One ordered stream has one head, and nothing in a segment says which application flow a byte belongs to. The measured case: a reply stalls for 3.01 s while 214 chunks belonging to three other flows cross the same connection, with zero retransmits (qqt benchmarks §1). That was measured on the QUIC tunnel. Move the same reply to the head of a TCP socket and the arithmetic follows, modelled from the same measured constants.

The single window plateaus at 2.0 MB, one BDP. The sender keeps offering at the rate the far side can absorb on a QUIC path, so the excess offered rate is 504 − 177 = 327 Mbit/s, or 41 MB/s. Two megabytes of buffer space therefore fills in 2.0 ÷ 41 ≈ 49 ms, before a human has noticed

the stall. After release the queue drains at 177 Mbit/s: 2.0 MB in 90.4 ms, exactly one round trip, and a full 4.0 MB clamp in 181 ms. Queueing delay is queue ÷ 177 Mbit/s, so every flow's measured round trip reads 90.4 ms empty, 180.8 ms at one BDP, and 271.2 ms at the clamp. The 2.6 MB reply itself fits 1.3 windows, so it needs two round trips, 180.8 ms, before its last byte lands.

The point is the sharing. A flow that lost nothing and wrote nothing still pays the stall plus the drain, because its bytes are behind the head in write order and the stream cannot skip ahead.

What the relay measured

Same file, same client, same moment, alternating transports: 177 Mbit/s through the relayed `ssh -L`, 504 Mbit/s through qqt over QUIC (benchmarks §1). The volume states that comparison as one ratio throughout, the tunnel's goodput over the relay's:

$$R = \frac{G_{\mathrm{qqt}}}{G_{\mathrm{ssh}}} = \frac{504\ \mathrm{Mbit/s}}{177\ \mathrm{Mbit/s}} \approx 2.85 \tag{6}$$

On a 3.0 GB file the QUIC panel finishes at 55.62 s; the TCP panel at 177 Mbit/s has delivered 2.48 GB at 120 s and is still running.

Under loss the gap stops being a ratio. In the emulated matrix at 90 ms RTT, `ssh -L` with eight streams delivers 0.0–3.7 Mbit/s at every lossy profile from 0.5 % upward, against 251 Mbit/s clean (benchmarks §2). The clean emulated ceiling is a different path from the relayed production ceiling, but the shape is the same: one shared window, every multiplexed connection head-of-line blocked behind one retransmit. The same matrix shows the controller decides how much of the collapse a single congestion domain suffers — qqt's own shared domain holds 336–367 Mbit/s at 1 % loss under BBR and falls to 2.5–8.6 under CUBIC. A second reliability layer is the structural cost; the controller is the multiplier on top.

Where the arithmetic stops

Degradation is conditional. Honda et al. found tunnels that "significantly improve" end-to-end performance under some conditions, and a loss repaired by fast retransmit is not a meltdown. No verified source gives a throughput multiplier for a loss rate: the 2× ladder is normative, the resulting slowdown is not a published number. The plateau, the 49 ms clamp, and the 4.0 MB buffer ceiling are model; the RTTs, the ceilings, the 3.01 s stall and the 214 chunks are measurement. And QUIC keeps a timer that doubles too (RFC 9002 §6.2.1). What it removes is the nested controller and the cross-stream head-of-line block (RFC 9000 §13), not timers.

---

03 · HEAD-OF-LINE BLOCKING

Head-of-line blocking: the wire carries no flow identity

A segment names a connection, not a flow

A TCP segment carries no field that says which application flow its payload belongs to. The header names a connection: address and port pair, sequence number, acknowledgement, flags, window (RFC 9293 §3.1). Everything above that is opaque bytes; a byte's only identity is its offset in the connection's sequence space.

TCP promises "a reliable, in-order, byte-stream service to applications" (RFC 9293 §2.2). A byte stream has a head, and delivery is in order. Write a chat token, then 1400 bytes of a file,

then a JSON-RPC reply into one socket and the receiver sees one sequence: offsets 0 through 11 are chat, 12 through 1411 are file. Nothing in the wire format records where one flow ends and the next begins. Four flows in one socket are four byte ranges in one queue.

The head is the whole argument

The receiver cannot hand offset 12 to the application until offset 11 has arrived, whatever flow it came from. The head stops for reasons that are not loss. A reader that stops calling `recv` stops it at the application: on mobile, QUIC spent 58% of its time application-limited against 7% on desktop, the process "unable to consume received packets as quickly as on a desktop" (Kakhki et al. 2017, §5.2). A lost segment stops it at the transport. In every case the blast radius is the connection, not the flow.

When the stop is a lost packet, the delay a byte behind it suffers has a floor set by the retransmission timer and by how many round trips the repair costs. A packet declared lost after the time threshold is retransmitted and acknowledged one round trip later, so a byte behind it waits at least the remaining round trip plus a full one:

$$d_{\mathrm{hol}} \;\geq\; R_{\mathrm{tt}} + (R_{\mathrm{tt}} + t_{\mathrm{reo}}) = 2R_{\mathrm{tt}} + t_{\mathrm{reo}} \tag{7}$$

where t_reo is the time the sender waited before it declared the packet lost — the time threshold or the packet threshold, whichever fired (RFC 9002 §6.1). On the house's relayed path at 90.4 ms that floor is 180.8 ms before any queueing is counted; the measured stall was 3.01 s, because the delay that reached the other flows was the queue's, not one repair's.

HTTP/2 made the argument one layer up

HTTP/1.1 put each request on its own TCP connection, so six parallel requests were six queues and one slow response blocked only itself. HTTP/2 multiplexed them onto one connection with stream identifiers in its framing layer, removing the application-layer queue but not the transport one: every stream still rode a single ordered byte stream. "While h2 eliminates HOL blocking on the application layer, it retains HOL blocking on the transport layer" (Goel et al. 2017).

Their control makes it a measurement: sharding large objects across four connections gave "13% faster PLTs" on a poor cellular path, 3% or nothing on good, fair and passable paths.

Keep the analogy honest in two places. HTTP/2 is evidence about an application multiplexing over one TCP connection; a tunnel is a different application of the same shape, so the result transfers as an argument about the transport, not a measurement of any tunnel. And HTTP/2's streams do carry identity: the framing layer writes a stream identifier into the payload, while TCP beneath it sees anonymous bytes. The multiplexing is visible to the endpoints and invisible to the transport, which is why the transport can block it.

What the shared window does with the four

The house scenario puts four flows on one link: 3.0 GB of bulk, 120 KB of chat, 240 KB of RPC, one reply that thinks for three seconds (mechanism note 07). On the relayed path the bulk ceiling measures 177 Mbit/s against QUIC's 504, and the shared window plateaus at 2.0 MB, 177 Mbit/s times the 90.4 ms path RTT.

The cold reply sits at the head for 3.01 s. The queue fills at the excess offered rate, 504 minus 177 Mbit/s, or 41 MB/s, so it clamps full 49 ms into the stall and drains in one bandwidth-delay product, 90.4 ms. RTT for every flow rises with the queue: 90.4, 180.8, 271.2 ms. Across the stall the three live flows hand over 214 chunks, 182 from bulk, 30 chat writes, 2 RPC replies, zero retransmits.

The run was QUIC-carried, so the cross-stream block never fired. What it shows is the other half: the shared queue exporting one flow's delay to all, while three flows that lost nothing waited.

The tunnel compounds it

A TCP tunnel writes every inner flow's bytes into one outer connection. For `ssh -L` this is literal: SSH multiplexes its channels over one TCP connection, so the outer sequence interleaves every forward's payload. A lost outer segment blocks every byte behind it, including unrelated inner flows' bytes (mechanism note 01, on RFC 9293 §2.2). The inner connections do not merely slow together; they lose their independence.

Three couplings follow.

- One window. The outer has one congestion window and one round-trip estimate; the inner flows each have their own. A loss belonging to flow A costs B, C and D the time the outer takes to deliver past it.
- One buffer. The tunnel process is a store-and-forward hop with a finite socket buffer. While the outer backs off, that buffer fills with bytes the inner already counts as in flight. Honda et al. (2005) and Harkanson et al. (2019) treat buffer sizing as a first-order term.
- One reader. The tunnel's copy loop is one reader of the outer stream. If an inner destination stops reading and its socket window closes, the write blocks, the loop stops draining the outer, and the outer window closes on every flow at once.

QUIC changes the scope of the first and third, not their existence. qqt gives each forwarded TCP connection its own stream, up to 1024 concurrent streams per connection (architecture_qqt-protocol). A loss blocks only the streams whose data it carried (RFC 9000 §13), though streams sharing a packet share its fate. Rate stays shared, because "the congestion controller is per path" (RFC 9002 §7), and budget stays shared, because connection-level flow control caps the total across all streams (RFC 9000 §4.1, §19.9). Loss can stall credit release too: "Packet loss can cause gaps in the receive buffer, preventing the application from consuming data and freeing up receive buffer space" (§4.3).

What the mechanism claims

For a plain TCP socket the claim is categorical, and no tuning removes it: a reader that stops at the head stops everything behind it.

For a tunnel it is conditional. The outer queue is shared however the inner flows are multiplexed, so a stall in one becomes queueing delay for all. QUIC removes the cross-stream delivery block, not the shared rate, budget or bottleneck queue. The HTTP/2 analogy relocates the block to the transport rather than deleting it, and the literature does not support "TCP-in-TCP is always catastrophic" (mechanism note 01, on Honda et al. 2005).

04 · WHAT QUIC ACTUALLY CHANGES

What QUIC actually changes: streams, flow control, and what it does not change

The claim under audit

"QUIC streams are independent, so one stream cannot hurt another." Half of that sentence is specification and half is wishful thinking, and RFC 9000 is explicit about which half is which. A claim stronger than the transport supports fails the first time a reader puts loss on the path.

An ordered byte sequence, and nothing else

RFC 9000 §2 defines the abstraction and bounds it in the same paragraph. Streams are a "lightweight, ordered byte-stream abstraction to an application"; they can be created by either endpoint, carry data concurrently with other streams, and be canceled. Then the sentence that does the work: "QUIC does not provide any means of ensuring ordering between bytes on different streams."

Within one stream, ordering is mandatory. Endpoints "MUST be able to deliver stream data to an application as an ordered byte stream", buffering out-of-order arrivals up to the advertised flow control limit (§2.2). Streams carry "no other structure visible to QUIC", and STREAM frame boundaries are not preserved across transmission, retransmission, or delivery. A stream is a position in a sequence, not a message.

The benefit is stated at §13, and so is its limit. When a packet is lost, "only streams with data in that packet are blocked waiting for a retransmission, while other streams can continue making progress." But: "when data from multiple streams is included in a single QUIC packet, loss of that packet blocks all those streams." Multiplexing is a packing decision, and packing re-couples what the stream abstraction separated. Independence is a property of delivery, not a promise that the wire will keep your flows apart.

The two things still shared

Congestion control is one window for every stream. RFC 9002 §7 makes the controller "per path", not per stream, and forbids sending when `bytes_in_flight` would exceed the congestion window. On the ordinary single-path connection that means one controller, one window, all streams. A loss anywhere halves it (§7.3.2), and persistent congestion collapses it further (§7.6). Loss detection is split across packet number spaces (§6), but that split is about keys, not flows.

Flow control is two limits stacked. RFC 9000 §4.1 describes a limit-based scheme with "two levels of data flow control": stream flow control, which stops one stream from consuming the whole receive buffer, and connection flow control, which stops senders from exceeding the receiver's buffer for the connection. "Senders MUST NOT send data in excess of either limit." The limits travel separately, `MAX_STREAM_DATA` per stream and `MAX_DATA` for the connection, and §19.9 closes the arithmetic: the sum of final sizes on all streams, terminal ones included, must not exceed the connection figure. One greedy stream can drain the shared budget and stall every other stream until credit is released.

There is a subtler coupling in §4.3. Packet loss creates gaps in the receive buffer, the application cannot consume past a gap, and credit release stalls for the whole connection. A loss on stream A throttles stream B through the credit path even though B's own bytes arrived.

The sentence that survives

QUIC gives an application stream-ordered delivery without cross-stream head-of-line blocking, plus per-stream flow control. It does not give per-stream congestion control, per-stream loss recovery, or a per-stream budget. Blocked delivery is scoped to a stream; rate and budget are not. That is the formulation qqt may print, and the one this chapter defends.

Mapping a forwarded connection onto a stream

qqt carries every forwarded TCP connection over a single mutual-TLS QUIC connection, one stream per flow: up to 1024 concurrent bidirectional streams per connection, in either initiation direction (quickquic, `architecture_qqt-protocol.md`). Each accepted TCP connection opens one bidirectional stream whose first byte is a frame type, followed by a dial header naming host and port. Whoever receives a data stream dials that header from its own network position. After the header the stream is a raw byte pipe in both directions.

A TCP FIN finishes the QUIC send stream, and the far side shuts down the TCP write side, so half-closing protocols behave as they do under `ssh -L`. An aborted pump resets the stream rather than ending it, because a peer that read a graceful FIN as EOF would truncate a file the protocol meant to fail.

The mapping is the clean one because it is nearly an identity. A TCP connection is already an ordered byte stream in each direction, with its own flow control, its own loss recovery, and its own FIN. QUIC offers exactly an ordered byte sequence in each direction of a bidirectional stream, with a flow control limit, a reset, and a FIN. Nothing has to be reframed: TCP does not preserve segment boundaries, and QUIC does not preserve STREAM frame boundaries, so neither layer is asked to carry a structure it does not have. The inner connection's end-to-end window stays its own; the stream window is a second brake in series, and the delivered rate is the minimum of the two, with `cwnd/RTT` and whatever the application reads.

The contrast is a tunnel that multiplexes many forwarded connections over one byte stream, as ssh does. There the inner connections become byte ranges in one sequence, so a single retransmission stalls all of them (quickquic, `rationale_qqt-benchmarks.md`). That is cross-channel head-of-line blocking created by the mapping, not by TCP.

What the mapping does not merge away

One QUIC connection means one congestion controller. A loss on the bulk flow halves the window the chat flow also rides, exactly as RFC 9002 §7 requires. The connection-level flow budget is shared for the same reason, and qqt's shipped defaults put a 4 MiB per-stream window under a 64 MiB connection window, so the per-stream limit is the binding one until many streams run at once (quickquic, `rationale_qqt-benchmarks.md`). The matrix shows the per-stream window as a real brake rather than a formality: on a clean 90 ms path a 1.25 MB window predicts 1.25 MB / 90 ms ≈ 111 Mbit/s against a measured 99, and raising it to 4 MB lifts the single-stream cell to 324.

The record also supplies a negative control. Across a 3.01 s first-token stall, 214 chunks flowed to other streams on the same connection with zero retransmits. That is what stream independence looks like when nothing is lost. It is evidence about the blocking claim, not about the controller, and it should never be read as the second.

05 · CONGESTION CONTROL IN THE TUNNEL

Congestion control in the tunnel: BBR, CUBIC, and what the house measured

A tunnel carries many flows and gets one controller. That mismatch is the whole subject, and where the interesting failures live.

One window per connection

QUIC gives each stream its own flow-control credit and its own ordering, which is why the protocol removes head-of-line blocking between streams. It does not give each stream its own congestion controller. RFC 9308 §12 states it without qualification: "QUIC, as defined in [QUIC], has a single congestion controller and recovery handler." RFC 9002 §7 puts that controller per path, and RFC 9000 §4.1 adds flow control on top as two levels — stream and connection. Flow control is the receiver's promise. Congestion control stays one number for the connection.

The consequence runs in five steps. The window is one byte count; each flow's entitlement is a slice of it. A loss anywhere cuts it everywhere, because the sender cannot tell which flow lost and takes `ssthresh` from the connection's flight size. The retransmission goes to the front of the queue and still counts as in flight (RFC 9002 §7), spending every other flow's credit. The receive window stops growing while the hole is open (RFC 9000 §4.3). And delivery is in order, so nothing behind the gap is readable — with QUIC narrowing this only to the packet: a packet carrying data from several streams blocks all of them (RFC 9000 §13).

So the plain statement, unsoftened: a loss halves one window for all flows, in TCP and still in QUIC. "Halves" is Reno's arithmetic, not a law — CUBIC cuts to 0.7 (RFC 9438 §3.4), and a timeout leaves one segment. The direction of the cut is what matters.

What BBR and CUBIC disagree about

CUBIC is loss-based. It grows the window on a cubic curve in elapsed time, concave toward the remembered pre-loss window and convex after it (RFC 9438 §4.2), and treats a loss as the signal to back off, respecting the invariant that a recovery round carries no more than half the segments outstanding when the loss was detected (§4.3). On a long fat path under sustained loss, the climb never finishes before the next cut.

BBR leaves the loss-based family. It estimates the bottleneck bandwidth and the propagation round trip and paces to their product; the window bounds the estimate rather than signalling it, and loss moves the rate down to the delivery rate actually observed on the first repair round (Cardwell et al. 2016). The measured difference on a 90 ms, 500 Mbit/s path at 1% loss is a cliff, not a preference: CUBIC's multiplicative decrease collapsed an eight-stream aggregate to 2.5–8.6 Mbit/s while BBR held 336–367 — a factor of 40 to 100 (congestion-control rationale, quoting the ten-cell matrix).

A tunnel sharpens the choice, because the outer connection's controller and the inner flows' controllers cannot see each other. An inner TCP flow backs off for a loss the outer connection may never observe, and the outer connection cuts its window for a loss the inner flow never hears about.

What the house measured: 60 MiB of window for 1.6 Mbit/s

The tunnel benchmark ran two qqt tunnels from mbp-m3 to blueview for five minutes each — one nested inside WireGuard, one direct to the public address — pushing 200 KB to an echo

server and back once a second. Three hundred bursts, zero failures, 61.44 MB each way on both paths.

Both ends logged qqt's stats every five seconds. The window reading is the finding:

TABLE 2. The published record's window reading against a 1.6 Mbit/s offered load. Two paths, one tunnel, five minutes each.

PATH	CLIENT CWND	SERVER CWND	APPLICATION OFFERED
`WireGuard`	61.0 MiB	61.3 MiB	1.6 Mbit/s
`Public`	60.0 MiB	60.1 MiB	1.6 Mbit/s

BBR inflated the congestion window to roughly 60 MiB while the application asked for 1.6 Mbit/s. The mismatch is measurable as a standing time: the window a controller holds, divided by the rate the application offers, is how long a byte waits in it before it leaves.

$$d_{\text{stand}} = \frac{W_{\text{cwnd}}}{R_{\text{offered}}} = \frac{61\ \text{MiB}}{1.6\ \text{Mbit/s}} \approx 320\ \text{s} \tag{8}$$

Five minutes and twenty seconds of buffer, held empty, on a household uplink. That is BBR probing for bandwidth nothing was requesting, and the consequence is not cosmetic: one qqt stream hoards buffer and takes far more than a fair share of a shared uplink. The manual's own note says bbrv1 takes roughly twice a fair share against CUBIC co-flows — the same reading that keeps ESnet from recommending bbrv1 on mixed networks — and that quinn's pacing derives from cwnd rather than from the bandwidth estimate, so nothing in the controller ties cwnd to the path.

The shipped connection-level receive window, capped at max(64 MiB, per-stream window), bounds that growth in practice: the 0.6.0 verification reads cwnd at 31–46 MB under the cap on clean cells where the unbounded-window experiment read 683–687 MB. That bound is flow-control credit, not a rate model.

Two honest limits. The run measured quality, not ceiling — the 200 KB/s offering was the limiter, so 1.6 Mbit/s is what was asked for, not what the tunnel can do. And the two paths ran sequentially, 15:05–15:10 and 15:10–15:15 UTC, so some of the variance gap could be time-varying congestion.

Silent loss in the receive direction

The benchmark's other finding is harder to fix. On the WireGuard path the server counted 229 lost packets (0.39%) while its own client counted 1. The public path showed the same asymmetry, smaller: 3 at the server against 1 at the client. The server also registered five congestion events (`cong=5`) against zero on its client.

A receiver's lost counter describes what it expected and did not get. When the server loses packets in its receive direction, the client — the sender for that direction — is told nothing. It sees no loss, keeps growing cwnd, and keeps sending into a hole. BBR cannot react to loss it never learns about, so its model of the path stays wrong indefinitely. The benchmark note ties this to the observed jitter gap, 285 µs on the nested path against 37–49 ms on the public one.

What to set

- Lossy path, fast or dedicated link: keep the default BBR. The measured gap under loss is a factor of 40–100.

- Shared uplink with household CUBIC co-flows: `--cc cubic`, eyes open. BBR's fairness cost is real there; CUBIC is steady and well-mannered.
- Slow uplink (≤100 Mbit/s): consider a smaller `--recv-window`. 4 MiB is up to 7× the bandwidth-delay product there, and a concurrent bulk transfer can park that much data ahead of token traffic.
- Host qdisc: `tc fq maxrate <path rate>` caps the UDP flow's bursts on the datacenter side — the one ESnet long-fat item that transfers to a userspace QUIC tunnel; plain `fq` does not pace UDP.

The structural fix is a rate pacer in quinn's transmit loop, bounding cwnd by construction, plus receive-window auto-tuning so the static knob retires. Neither is qqt's to land alone.

---

06 · WINDOWS, BUFFERS AND LOSS

Windows, buffers and loss: the arithmetic a deployer has to do

A link's capacity is a rate. What a socket puts on it is a length divided by a time.

The product, stated plainly

The bandwidth-delay product is the number of bytes that must be in flight for a path to be full:

```
BDP = bandwidth × RTT
```

ESnet's tuning manual works the law: "if your ping time is 50 ms … the TCP buffers should be: .05 sec * (1 Gbit / 8 bits) = 6.25 MBytes" (fasterdata, `host-tuning/background`). It states the failure mode on the same page, and it is the sentence a tunnel operator should memorize: "The maximum congestion window is related to the amount of buffer space that the kernel allocates for each socket … If the buffers are too small, the TCP congestion window will never fully open up."

The same manual declines to pretend one stream fills a very fat pipe: "a 10Gbps flow across a 100ms network requires 120MB of buffering … Most data movement applications, such as Globus, would employ 4-8 streams" (fasterdata, `host-tuning/linux`). Window scaling lifts the 16-bit header field to 30 bits. It does not lift the buffer.

The binding window is a `min()` of two. RFC 5681 §2 forbids sending beyond "the sum of the highest acknowledged sequence number and the minimum of cwnd and rwnd". QUIC keeps one congestion window and adds "two levels of data flow control", stream and connection, on top (RFC 9000 §4.1). The arithmetic is unchanged; only the field name is.

The 2.0 MB plateau

The house's production path is 41.5 ms direct and 90.4 ms through the platform's relay, measured (qqt benchmark record §1). On the relayed path, `ssh -L` bulk transfer plateaus at 177 Mbit/s, and the socket's own statistics show a congestion window of 2.0 MB at that plateau. Multiply it out:

$$W_{\mathrm{bdp}} = 177\ \mathrm{Mbit/s} \times 90.4\ \mathrm{ms} = 16.0\ \mathrm{Mbit} = 2.0\ \mathrm{MB} \tag{9}$$

That is not a coincidence, and not a buffer shortage. It is `rate = window / RTT` read backwards: the socket holds one path's worth of bytes and turns them over once per round trip. On the direct path the same law produces 2.6 MB at 504 Mbit/s × 41.5 ms, and the record again measured it.

The queue around the plateau is what a deployer feels. Standing bytes in a queue ahead of a link delay every byte behind them by the queue divided by the link rate:

$$d_{\mathrm{q}} = \frac{Q}{C} \tag{10}$$

One BDP standing in the send queue costs exactly one RTT of delay, and the modelled 4.0 MB clamp costs two: the 90.4 ms path presents itself to an application as 180.8 ms and then 271.2 ms. Worked at the modelled clamp on the emulated 100 Mbit/s link, a 4 MiB buffer holds the path for a third of a second:

$$d_{\mathrm{q}} = \frac{4\ \mathrm{MiB}}{100\ \mathrm{Mbit/s}} = \frac{4{,}194{,}304\ \mathrm{B}}{12.5 \times 10^{6}\ \mathrm{B/s}} \approx 336\ \mathrm{ms} \tag{11}$$

A one-second-old connection carries 238.6 KiB, 0.24 MB (qqt benchmark record §4), so the window is small exactly when a first request needs it.

Why the ceiling belongs to the path

The tempting reading of 177 Mbit/s is that the tunnel is slow. The record rules that out by measuring the alternative: a QUIC tunnel on the same class of path reaches 504 Mbit/s. Multiplexing adds contention inside one window and never adds capacity. The tunnel chooses how much of the path it occupies; it does not choose the path.

QUIC relocates the constraint into a receive window with the same shape, and the record's window matrix traces the curve: at 90 ms and 500 Mbit/s, a BDP of about 5.6 MB, a 1.25 MB per-stream window delivered 99 Mbit/s, 4 MB delivered 324, and 8 MB delivered 474. That is 22% of BDP turning into roughly 20% of the pipe, and 143% into roughly 95% (qqt benchmark record §3).

The acknowledgment path is a path

One direction is the easy case. Load both and a second constraint appears on the return path, carrying no payload at all.

At 504 Mbit/s and roughly 1,200 bytes of payload per packet, the flow is 52,500 packets per second. One acknowledgment per two packets is 26,250 acks per second, and at about 66 bytes on the wire that is 1.7 MB/s, or 14 Mbit/s of reverse traffic carrying nothing — 3.5% of the upload's own rate, on its queue (scenario model, from the benchmark record).

Now let the client's send queue stand at 4 MB. Those acks wait 4.0 MB ÷ 395 Mbit/s = 81 ms. The download's round trip goes from 41.5 ms to 122 ms while the downlink itself is idle, and delivery falls from 504 to 170 Mbit/s. The slow direction throttles the fast one by delaying acknowledgments (RFC 3449 §3.1). No downlink buffer fixes it: the missing resource is queue on the other side of the connection.

When loss cuts the window

Loss shrinks the same window, and the size of the cut is version-specific: half under Reno and NewReno, 0.7 under CUBIC (RFC 9438 §3.4), one segment on a timeout. ESnet measured the consequence on a 90 ms path: at 0.0046% loss, 490 Mbit/s against 8.2 Gbit/s lossless, "almost

17 times slower" (fasterdata, `tcp-issues-explained/packet-loss`). The house's matrix is blunter. At 1% loss with eight streams, qqt on bbr held 336–367 Mbit/s and qqt on cubic fell to 2.5–8.6: a 40–130× gap from the controller alone (qqt benchmark record §2, §3). Cubic's number is not a lane running slowly. It is a window that fails to re-open, exactly the failure fasterdata's per-socket sentence predicts.

The deployer's rule

Size buffers to **`bandwidth × RTT`** for the rate you actually run, and measure before you change anything. On the production path class that is 500 Mbit/s × 90 ms ≈ 5.6 MB. Linux's stock `net.core.rmem_default` is 212,992 bytes, about 208 KiB, roughly 27× smaller (qqt tuning). qqt never calls `setsockopt`, so the `_default` *sysctls are the ones it rides;* `_max` alone changes nothing for it. Raising `*_default` changes every new socket on the host, so scope it to a network namespace if you can.

The measurement caveat is the reason this page says *measure*. On the emulated 500 Mbit/s, 90 ms path, stock buffers, `_max` *at 4 MiB with defaults stock, and* `_max` plus `*_default` at 4 MiB delivered 471/474, 470/475, and 470/472 Mbit/s: no measurable difference, because quinn sets no socket buffer and GSO/GRO coalescing with a draining reader held a 5.6 MB BDP on 208 KiB (qqt tuning). Unshaped, the same pair ran at 5.2–7.6 Gbit/s and the receivers' `RcvbufErrors` moved: 5,312 on one cell, 304,428 on another. The ceiling is rate-dependent and window-dependent: at an 8 MiB window, cubic's clean-path aggregate collapsed to 57.7 Mbit/s where 4 MiB delivered 453, with the sender's loss count and the receiver's `RcvbufErrors` pinning the cause (qqt congestion record). A window is a burst license as much as a throughput license.

Keep **`--cc bbr`** unless you have a reason on paper. It is the default, and the default is the matrix's adjudication, not an inheritance. Choose cubic in one case: a shared uplink with household cubic co-flows, where bbrv1 takes roughly twice its fair share, eyes open. On a slow uplink of 100 Mbit/s or less, consider a smaller `--recv-window`; the shipped 4 MiB is up to 7× the BDP there, and a concurrent bulk transfer parks that much data ahead of token traffic. Neither flag has a kernel-side consequence, and a change to either without a re-run is unpriced.

---

07 · SETUP COST

Setup cost: handshake, 0-RTT, and reconnection

The handshake, counted in round trips

A reader waiting on the first byte pays two costs: a handshake, fixed in round trips, and a congestion-window ramp, priced by the path's bandwidth-delay product. On a long path the ramp is the larger bill. A tunnel's claim is that it buys the handshake once, and that the ramp belongs to the tunnel rather than to each flow.

QUIC carries TLS 1.3 as its handshake layer, and a full handshake costs one round trip that establishes keys, transport parameters and authentication together (RFC 9001). qqt requires mutual certificates inside it: the server admits the client's chain from its trust bundle, the client verifies the server's `quic-tun` SAN, and the ALPN must match (`architecture_qqt-protocol.md` §Transport and handshake). The revocation deny-list is re-read at every handshake.

TLS over TCP splits this: TCP's round trip, then TLS's. Naylor et al. (2014, §2) count two round trips for a TLS handshake against TCP's one. TLS 1.3 removes one of those, not the TCP one, so TCP plus TLS 1.3 still costs two.

TABLE 3. Round trips before the first payload byte. The counts are structural; the times below are arithmetic on the measured path.

TRANSPORT	ROUND TRIPS BEFORE FIRST BYTE	NOTE
`TCP + TLS 1.2`	3	two TLS round trips over TCP's one (Naylor et al. 2014)
`TCP + TLS 1.3`	2	one TCP, one TLS
`QUIC full handshake`	1	transport and TLS together (RFC 9001)
`QUIC resumed, 0-RTT`	0	replayable (RFC 9001 §4.6)

The saving is one round trip per connection (`quic-advantage-mechanism/08`). Written as time, the handshake a reader waits out before the first byte of payload is the round-trip count times the path's round trip:

$$d_{\mathrm{hs}} = n_{\mathrm{rtt}} \times R_{\mathrm{tt}}: \quad 2 \times 41.5\ \mathrm{ms} = 83.0\ \mathrm{ms}\ \text{against}\ 1 \times 41.5\ \mathrm{ms} = 41.5\ \mathrm{ms} \tag{12}$$

On the production path that round trip is 41.5 ms against the relayed path's 90.4 ms, and each relayed trip carries a 48.9 ms premium.

0-RTT, and why the reconnect is not it

TLS 1.3 offers resumption, and QUIC carries the zero-round-trip form: a client remembering a prior session sends application data in its first flight (RFC 9001 §4.6). The data is replayable, since an observer can resend the flight and the server accepts it, and before address validation the server is amplification-limited (RFC 9002 §7.2). Zero-RTT suits a request safe to repeat, not an action that must happen once.

qqt does not expose it. No early-data mode appears in the wire contract, and the lifecycle is explicit: with `--reconnect` the client re-resolves and re-dials, and "a reconnect is a fresh handshake and fresh streams, never a transparent takeover; in-flight streams die with the old connection" (`architecture_qqt-protocol.md` §Connection lifecycle). A redial costs one full round trip and a certificate exchange. The conservative reading is that the replayable payload here would be a dial header, and a replayed dial header is a second connection to the target.

The ramp is the recurring bill

Every QUIC connection begins in slow start, with an initial window of ten datagrams capped at 14,720 bytes (RFC 9002 §7.2); TCP starts at 14,600 bytes (RFC 6928). The controller is per path, so a new stream on a warm connection arrives with a window somebody else grew and ramps not at all (`quic-advantage-mechanism/04`). Slow start doubles the window each round trip, so the number of round trips to reach a path's bandwidth-delay product from the initial window is a logarithm:

$$n_{\mathrm{rtts}} = \left\lceil \log_2 \frac{W_{\mathrm{bdp}}}{W_{\mathrm{init}}} \right\rceil = \left\lceil \log_2 \frac{500\ \mathrm{kB}}{14.72\ \mathrm{kB}} \right\rceil = \lceil 5.09 \rceil = 6 \tag{13}$$

Filling a 500 KB bandwidth-delay product therefore takes six round trips, about 240 ms on a 40 ms path; 5 MB takes nine, about 360 ms, or 0.6 % of a 40-second transfer. A 4 KB request fits the initial window and pays round trips, not growth.

A fresh TCP connection pays the ramp every time; a tunnel pays it once. That is the mechanism behind the house numbers, and it cuts both ways.

What the house measured

The production record alternates identical requests through a relayed `ssh -L` tunnel and through qqt, same backend (`rationale_qqt-benchmarks.md` §1). Cold, with a new channel per request, the relayed path costs 539.5 ms against qqt's 50.7 ms, a 10.6× ratio. Warm, 240.0 against 47.7, or 5.0×. qqt's cold-to-warm gap is about 3 ms: the cost of a fresh channel inside an established tunnel. The relayed gap is about 300 ms: a cold relayed channel redoes the relay's setup.

Stream setup is near-free: −0.3 and −0.8 ms in the size-matched test, and no handshake appears in a request's budget. A 120 s capture carried zero QUIC handshakes across 117 requests, with 15 TCP SYNs and 6.8 to 9.3 requests per connection.

What an established tunnel does that a fresh connection cannot

A new forwarded connection opens a stream and inherits a window minutes in the growing. It skips slow start and the handshake, so the first byte leaves after one path round trip plus the server's dial. Five tests at up to seven streams found no head-of-line blocking, and during a 3.01 s first-token stall 214 chunks flowed to other streams with zero retransmits.

The inherited window is not the stream's own: one window, one loss-recovery context, one round-trip estimate (RFC 9000 §2, §4.1; RFC 9002 §7). A flow opening during bulk traffic enters a queue it did not create, and a bulk-flow loss shrinks the window the interactive flow rides. A fresh TCP connection would have started clean.

The window can also grow past what the traffic asks. In the two-path benchmark (`qqt-tunnel-benchmark/index.md`) both tunnels inflated `cwnd` to about 60 MiB while the application offered 1.6 Mbit/s; one qqt stream hoards buffer, and BBR takes roughly twice a fair share against cubic co-flows. `--cc cubic` is the polite choice on shared uplinks.

On a clean path, single stream, the mechanism note's harness measured qqt at 100/102 Mbit/s against raw TCP's 337/336, so the per-packet charge is real.

Reconnection: what the tunnel does when the path dies

Loss is detected by the 60 s idle timeout, a 10 s client keepalive holding a live path open, so a hard crash costs up to 60 s of black hole (`instructions_qqt-howto.md` §6). Without `--reconnect` the client exits 1 and supervision retries, shipped as `Restart=always` with `RestartSec=2`. With it, the client walks the resolver's admitted list, one `CONNECT_TIMEOUT` per candidate, on fixed 2 s pacing rather than remembering a winner (`rationale_qqt-address-family.md`).

A cold server is the rental-platform case: the client exits and reconnects every 2 s until the server answers, and the howto carries an end-to-end measurement of a platform reboot. Reverse listeners leave the one mark across the gap: a redial re-asks for the binds, the port is closed while the tunnel is down, and a bind still held by the dead connection's idle timer is retried across a roughly 70 s budget.

Certificates land here too. An expired leaf stops admitting on its own, so a stale-cert client fails at the redial rather than degrading; rollover means minting the leaf under the customer's CA, swapping the cert directory, and restarting the client. The deny-list bites at the same moment, and a live connection keeps its streams until it drops.

What the reader is buying

One round trip instead of two, once per tunnel. Then a warm window every forwarded flow enters without a ramp, worth tens of milliseconds per cold channel on a 41.5 ms path and the difference between 50.7 and 539.5 ms in the record. The bill is shared fate: one window, its loss everyone's, and a tunnel left to BBR holding more of a shared uplink than its traffic justifies.

---

08 · WHAT THE PUBLISHED RECORD SAYS (AND WHAT IT DOES NOT)

What the published record says (and what it does not)

The folklore is one sentence: QUIC is faster than TCP. The literature is a set of conditional results. Its authors agree on when the gain appears — long round-trip time, real loss, multiple objects competing — and agree on what happens when those are taken away. Take them away and the median gain falls toward zero. Take them away completely and, on a fast enough clean path, it changes sign.

The shape, in two numbers

Google ran an A/B test on production traffic: QUIC against a TLS/TCP control, measuring Search, YouTube playback and rebuffer rate (Langley et al. 2017). The headline means are an 8.0 % reduction in Search latency for desktop users and 3.6 % for mobile users, and rebuffer-rate reductions of 18.0 % and 15.3 % respectively. The means are the least interesting numbers in the paper. Broken out by percentile, desktop Search improves 1.5 % at the 50th, 5.8 % at the 90th and 16.7 % at the 99th. Mobile Search improves 0.5 % at the 50th — and −0.6 % at the 1st, meaning QUIC was slower on the fastest mobile connections. The authors' own sentence is the one to keep: "QUIC's improvements increase as base Search Latency increases."

Wolsing et al. (2019) supply the other end, and did the hard thing: their baseline is TCP*tuned to QUIC's own defaults* — initial window of 32 segments, pacing, no slow start after idle, tuned buffers. Against that baseline, over four emulated networks scored on visual metrics rather than page-load time, QUIC wins −87.1 ms of Speed Index on a clean 24 ms DSL link with 0 % loss, and −215.9 ms on a 74 ms LTE link at 0 % loss. On a 760 ms link carrying 6.0 % loss the same comparison is −8.36 s. Two orders of magnitude separate the clean link from the lossy one, with no change of protocol or method. Their attribution: the advantage is "mostly caused by QUIC's reduced RTT design during connection establishment, and, in case of lossy networks due to its ability to circumvent head-of-line blocking."

That is the pattern. Everything else in the corpus confirms it at another scale, explains the mechanism, or marks its boundary.

Confirmation at scale

Trevisan et al. (2021) measured 14,707 websites over 2,647,260 visits across a month, HTTP/3 against HTTP/2, imposing latency, loss and bandwidth one factor at a time on a clean two-server campus baseline. On the native path the result is a coin flip: "approximately in 50 % of the cases websites load faster with HTTP/3 and in the remaining cases HTTP/3 is slower." Add 50 ms and 69 % of sites load faster (74 % on Speed Index); at +100 ms, 76 % (81 %); at +200 ms, 81 % (87 %). Their summary restates it: "HTTP/3 provides sizable benefits only in scenarios with high latency or poor bandwidth."

Kakhki et al. (2017) replayed roughly 500 real pages across desktop, mobile, wired, cellular and proxied paths, and found QUIC outperforming TCP+HTTPS "in nearly every scenario" on desktop — with two named failure modes. QUIC was sensitive to out-of-order delivery, performing "significantly worse than TCP in many scenarios" when packets were reordered, and "all of QUIC's performance gains are diminished on phones from 2013 and late 2014." Both are cautions about implementation and hardware age, not the design.

Where the gain actually comes from

Two studies separate mechanism from marketing.

Goel et al. (2017) is the cleanest measurement of the head-of-line claim, and it is about HTTP/2, not QUIC: "While h2 eliminates HOL blocking on the application layer, it retains HOL blocking on the transport layer." Six TCP connections against one HTTP/2 connection is the control that proves which layer bites. Spreading large objects over four connections bought 3 % or nothing on Good, Fair and Passable paths, but 13 % faster page loads on a Poor path. On an 8 MB page under Good conditions, a single connection was 53 % faster than six. The blocking moves; it does not vanish.

Honda et al. (2005) measured the tunnelling case directly, fifteen years before QUIC, in simulation with TCP Reno: end-to-end goodput and RTT for a TCP flow carried inside a TCP tunnel, across a matrix of access delay, backbone delay, SACK state and buffer sizing. The result cuts both ways. "When both the propagation delay … of the access link and the propagation delay … of the backbone link are large, using the TCP tunnel increases the goodput of the TCP flow by approximately 1.5 times." Otherwise "using a TCP tunnel slightly decreases the goodput of a TCP flow." RTT always pays: it "increases … approximately 1.4 to 3 times." TCP-in-TCP is a regime, not a law.

The other half of the curve

Zhang et al. (2024) ran QUIC+HTTP/3 against TCP+TLS+HTTP/2 over wired Ethernet, Wi-Fi 6/7 and 5G, from 100 Mbit/s to 1 Gbit/s. "Over fast Internet, the UDP+QUIC+HTTP/3 stack suffers a data rate reduction of up to 45.2 %." The crossover is explicit at about 500 Mbit/s; at 1 Gbit/s, on one file over desktop Ethernet, 9.32 s became 18.60 s. The cause is receiver-side processing — excessive packets and QUIC's user-space ACKs, with no UDP GRO to match TCP's.

One counterweight belongs beside it. Seufert et al. (2019) streamed YouTube over both transports on two mobile and two fixed networks and found "no evidence for any QoE improvement of QUIC over TCP" — the application had already removed the waiting QUIC saves. It does not contradict Langley et al.; it is the same mechanism, pre-empted one layer up.

What the literature does not say

Three limits bound everything above.

First, the traffic is not tunnel traffic. Six of the seven measure web pages, video or bulk transfer in a browser; the seventh measures a simulated TCP tunnel. Not one measured a mutually authenticated QUIC tunnel carrying SSH sessions between two of the owner's hosts. A Web page's object graph and an interactive shell's small request-response cycles share a transport but not a workload.

Second, tuned-versus-untuned TCP is the standing confound. Wolsing et al. show the gap shrinking by an order of magnitude when the TCP baseline is given QUIC's own defaults. A study that compares QUIC to a default Linux stack has measured the tuning gap as much as the protocol. Where a paper states its baseline, read it before the effect size.

Third, significance is not magnitude. Wolsing's QUIC is significantly faster on a 0 %-loss DSL link — by 87 ms, invisible to a user. And a visual metric can invert outright: on nytimes.com, TCP reached VC85 at about 48 s and QUIC at about 124 s, while QUIC finished the page sooner. "QUIC wins" without a path, a percentile and a baseline is not a claim the literature supports.

That is why the house ran its own: 41.5 ms and 90.4 ms path RTTs, a 3.01 s cold stall, 214 chunks handed over it with zero retransmits. It is not a headline. With nothing lost, stream independence never fires — a good baseline, and a poor speed claim.

09 · HOUSE MEASUREMENT I

House measurement I: two paths, one tunnel

What a tunnel over UDP actually tests

qqt carries TCP inside one mutually authenticated QUIC connection, with one stream per forwarded connection. QUIC runs on UDP. So a qqt tunnel is a UDP flow, and every property a path has against UDP shows up in it first.

That matters here because the same tunnel can be handed two very different paths. The question this run asked was narrow: does nesting a QUIC tunnel inside WireGuard cost anything, and does the public path carry a defect that only the tunnel's own counters can show? (tunnel benchmark note)

The setup, stated precisely

Two tunnels ran from mbp-m3 to blueview, deliberately not nested inside each other. The WG path was a qqt server on `10.10.0.57:18443`, reached as QUIC inside `wg0`, a tunnel within a tunnel. The public path was a qqt server on `37.27.134.57:18445`, one layer, QUIC direct to the public address. Certificates were minted fresh with `qqt gencert`, servers and clients both ran `--stats 5`, and the Linux binary came from the miramore checkout because the macOS build does not execute on Linux.

The 63-byte MTU difference is the proof the paths were genuinely different. The nested path reported 1389, since the outer WireGuard encapsulation eats the headroom. The public path reported 1452. Had both reported the same MTU, the premise of the comparison would have been false.

The load was a byte-for-byte echo server on blueview: 200 KB pushed up, 200 KB returning, once per second, 300 times, five minutes. Both runs held one TCP connection for the whole window, so the numbers are steady state rather than connection setup. (tunnel benchmark note)

The application result

Both paths worked. 300 of 300 bursts completed on each, zero failures, 61.44 MB sent and 61.44 MB received in both directions. Round-trip goodput was 1.6 Mbit/s on both, which is exactly the 200 KB/s the client offered.

	WG PATH	PUBLIC PATH
Bursts ok	300 / 300	300 / 300
Median burst rtt	74.1 ms	116.1 ms
p95	129.8 ms	232.4 ms
max	245.7 ms	504.5 ms
Mean burst time	84.6 ms	131.6 ms

Both paths carried the same application result, with very different transport behaviour behind it.

The jitter separation

Both paths shared a 56 ms RTT floor. The stats read `rtt_min=56.1` on the WG client and `56.0` on its server, `56.5` and `56.3` on the public pair. What separates them is stability.

The WG path held 285 µs of jitter at the client and 0 µs at the server. The public path held 41.7 ms at the client and 37.4 ms at the server. That is roughly a 150-fold difference, and it appears independently at both ends of each tunnel, which rules out a single bad measurement point.

The plausible reading is that the WG path is a stable, low-loss tunnel between two known endpoints, so the outer WireGuard has already done the hard work of holding a clean path, while the public path inherits whatever the internet does with a UDP flow to Helsinki. (tunnel benchmark note) That is an inference, not a measurement, and the note labels it as one.

The latency distribution

Jitter at the transport layer becomes latency at the application layer. Median per-burst round trip was 74.1 ms on the WG path against 116.1 ms on the public path, roughly a 3x difference waited out on every single burst. p95 was 129.8 ms against 232.4 ms, roughly 2x. The tail was 245.7 ms against 504.5 ms.

Mean burst time, which includes the whole up-and-back exchange rather than the transport's own round-trip estimate, was 84.6 ms against 131.6 ms.

Five hundred milliseconds is a tail a human notices without instrumentation.

The loss disagreement is the finding

Each end of each tunnel kept its own counters. They did not agree.

On the WG path the client reported `lost=1 loss=0.00%` and the server reported `lost=229 loss=0.39%`. On the public path the client reported `lost=1` and the server reported `lost=3`, 0.01%. The WG server also registered `cong=5` against its client's `cong=1`.

A receiver's counters describe what it expected and did not get. Silent loss in the server's receive direction is invisible to the sender, so the sender's congestion controller sees no loss, keeps growing its window, and keeps sending into a hole. That is the mechanism that turns silent loss into exactly the jitter the public path shows and the nested path does not. (tunnel benchmark note)

The measurement is only possible because qqt reports stats on both ends from the same snapshot of live state. `Metrics::snapshot` reads the tunnel's state once and both renderings, the log line and the Prometheus body, format that one `Snap`, so the two surfaces cannot disagree

about what the tunnel is doing. Two ends of one connection, each printing its own view, is what makes a directional disagreement legible at all. (qqt stats)

Sixty mebibytes against 1.6 Mbit/s

Every stats line in the run showed a congestion window near 60 MiB: 61.0 and 61.3 on the WG pair, 60.0 and 60.1 on the public pair. The application was offering 1.6 Mbit/s, about 0.2 MB/s.

BBR was probing for bandwidth nothing was asking for. The practical consequence is that one qqt stream hoards far more buffer than its share, and the crate's own `--cc` note records that BBR takes roughly twice a fair share against CUBIC co-flows. On a shared uplink the polite choice is `--cc cubic`; BBR is right when the path is yours. The same window behaviour is measured again in the emulated matrix, where the shipped 64 MiB connection window is what bounds BBR's growth rather than the pacer. (qqt benchmarks §3)

Quality, not ceiling

The honest limit of this run is that 200 KB/s was the limiter. 1.6 Mbit/s is what was asked for, not what the tunnel can do. Finding a ceiling needs a saturated blast, which is a different run.

Two further caveats belong on the record. The runs were sequential, WG from 15:05 to 15:10 UTC and public from 15:10 to 15:15 UTC, so some of the variance gap could be time-varying congestion rather than the path itself; a simultaneous A/B run removes that confound. And this measured a path that never crossed the gateway's routed path, which is why it held steady where the house's UDP measurements on that path did not. (tunnel benchmark note)

What survives all three caveats is the shape of the result: identical RTT floor, a large stability gap, a latency tail halved, and a counter disagreement that no application-level check would ever have surfaced.

---

10 · HOUSE MEASUREMENT II

House measurement II: keystroke latency against a provider relay

The question this run asks

Throughput numbers tell you how long a file takes. They say nothing about how a terminal feels. A shell is a round-trip machine: every key you press is a byte out and a glyph back, and the clock between them is the difference between typing and waiting.

So the run poses one question. Hold the far end, the shell, the terminal and the keyboard constant, change only the route, and ask how much of the lag a typist feels belongs to the transport.

The design is an A/B with nothing else in it. One client, one keyboard, one keystroke stream, delivered into two terminals at once. The left pane reaches the box over a qqt tunnel. The right pane reaches the same box, running the same `sshd`, over the provider's SSH relay. Both panes receive the identical bytes in the identical instant (House keystroke note, §A tunnel against a relay). Every difference on screen is the way there.

The mechanism, before the numbers

The two paths differ in hop count, and hop count is what a keystroke pays for.

The tunnel is one QUIC connection from the client to the box's UDP port. From the measuring vantage the round trip on that path was 40.7 ms, and a keystroke's echo costs one round trip

plus a few milliseconds of terminal handling on the far side (House keystroke note, §Why the gap exists).

The relay is a store-and-forward hop. The client talks to the relay, the relay talks to the box, and each leg is its own round trip. Every keystroke pays roughly four hops instead of two. It adds a second conversation to one that was already latency-bound.

That predicts a ratio near two on hop count, and worse in practice, because each added leg carries its own queueing.

The harness

Transport order was alternated in blocks, not randomized per sample, so a drifting path moves both columns together instead of loading one side of the comparison.

Each cell holds 120 samples per transport, with zero rejections: every sample the harness started produced a usable frame pair, so no cell is a survivor of a filtered set. Four workloads across two transports give 1,440 retained samples (House keystroke note, §The measured gap; `summary.tsv`).

The numbers

Medians, ninetieth and ninety-ninth percentiles, in milliseconds.

WORKLOAD	QQT P50 / P90 / P99	RAW SSH P50 / P90 / P99
`single keystroke echo`	49.7 / 59.1 / 61.4	233.1 / 234.4 / 261.2
`vi page scroll (full-screen repaint)`	49.1 / 56.1 / 61.2	268.9 / 270.5 / 280.0
`burst, first glyph`	72.9 / 73.0 / 73.7	237.0 / 240.1 / 317.9
`burst, last glyph`	122.3 / 127.6 / 129.5	325.3 / 327.8 / 357.4

What each row is

Single keystroke echo. One key, one glyph. The smallest thing a typist can notice, and the thing every other row is built from. 49.7 against 233.1 ms.

`vi` page scroll. A real `Ctrl-F` page-down over a 500-line file. A sample counts as complete only when the full 2,106-byte screen repaint has arrived, not when its first byte has, because timing the paint is what a reader of the screen experiences. 49.1 against 268.9 ms.

Burst, first glyph. Twelve characters typed six milliseconds apart, at machine speed rather than human speed. The first glyph is when the echo starts. 72.9 against 237.0 ms.

Burst, last glyph. This is the row a typist should read first. The first glyph tells you the far end woke up; the last glyph tells you the word you typed is actually in front of you. Nobody watching a burst asks about the second letter. The question is "has my word landed yet", and the twelfth glyph answers it. It is the interval a typist actually watches (House keystroke note, §The measured gap). 122.3 against 325.3 ms.

The gap between the two burst rows is itself informative. The word completes about 49 ms after it starts arriving on the tunnel, roughly 88 ms on the relay. A slow path also spreads: the relay's burst last-glyph p99 reaches 357.4 ms against the tunnel's 129.5 ms.

How honest the clock is

The measurement is not a packet capture with a synthetic timestamp. It is optical. The recording runs at 60 frames per second, and the clock at the centre of each clip is drawn in the framebuffer

itself, so the timestamp on any frozen frame is that frame's own. The keypress is a live overlay band at the bottom of the frame, so the instant of the keystroke is on screen as well as in the terminal (House keystroke note, §Watch it).

Reading the frames independently of the harness reproduces the table's shape: an on-screen p50 and p90 of 50 ms and 50 ms for the tunnel pane, 217 ms and 283 ms for the relay pane (House keystroke note, §The same lag, measured off the frames). The clip and the sample file agree because they measure one event.

The limit of this clock is 16.7 ms of frame quantization, one frame at 60 fps. The accurate reading of the tunnel column is therefore "one to three frames", not "49.7 ms exactly". A median inside 50 ms cannot be resolved further, and the run does not pretend otherwise. The relay column sits far above that floor, which is why the ratio survives a limit the tunnel's absolute figure does not.

What the ratio does and does not claim

The finding is the ratio, a bit under five to one on keystroke echo. The absolute numbers belong to the path. This client sat 40.7 ms from the far end. Sit closer and both columns shrink. Route through a worse relay and the right column grows. Anyone reproducing this should expect the same shape and their own two medians.

The far end was a container on a rented GPU host, lightly loaded during the run; the client was a Linux VM. Both panes ran the same `bash`, so shell, terminal and X server are common-mode and cancel out.

The same rig run on a directly reachable EC2 instance, where both paths measured a 164 ms round trip, produced a far smaller difference: 170.8 against 179.9 ms on echo, 212.7 against 237.8 ms on the burst's first glyph (House keystroke note, §What the same-RTT comparison says). Nine milliseconds on a key, twenty-five on typeahead. That pairing is the control: where the far end is directly reachable, a tunnel wins on throughput and typeahead, not on the single-key round trip, because the round trip is the speed of light on both paths. The five-to-one appears when a relay stands between the typist and the box.

---

11 · THE 14 MB DROP

The 14 MB drop: one transfer, diagnosed

The order, and the rehearsal

The question was not whether qqt could carry a binary. It was whether `ssh` earned its place in a ship step at all. After a deploy that used `ssh 'cat >'` to land a rebuilt host-control-api binary on a vast cell, the owner asked the sharper question: why ssh, why not just pipe the bytes (the drop note, 1 November 2026, 15:18:42).

The first answer was a rehearsal. A python3 one-shot listener on the box, a qqt forward, and 300,000 bytes pushed straight through with no ssh, no scp and no key. Both ends hashed to `f5d9de99143fa2b3bd9b1bef6f94d13f7a52c225657064ef176410f74a09ee16`. That is a fact, recorded in the transcript. It establishes carriage and nothing about rate.

Then the owner said four words: *time it?*

Four readings of one transfer

The measured object was the real artifact: 14,505,488 bytes, shipped to the same box three ways, every delivery hash-verified.

READING	REPORTED	IMPLIED RATE	DISPOSITION
`raw qqt, attempt 1`	0.04 s	~362 MB/s	false clock
`raw qqt, attempt 2`	2.87 s	~5.1 MB/s	true but harness-damaged
`raw qqt, final`	0.76 s	~19 MB/s	the published number
`ssh cat >`	2.1-2.5 s	~6.5 MB/s	reference
`scp`	3.7-4.0 s	~3.8 MB/s	reference

(Five rows, one object; the note's own table prints the last three.)

The same tunnel, the same file, the same two hosts: 0.04 s, then 2.87 s, then 0.76 s. A factor of seventy between the extremes. No tuning was applied between them. Nothing on the wire changed. That spread is the incident.

Where the clock stopped

Fact, stated by the source: the 0.04 s figure was wrong, and the transcript says why in the agent's own words. `sendall` returns once the bytes are in the kernel send buffer, not when they land. The transfer was complete and correct, full 14,505,488 bytes, hash matched. The timer simply stopped at the wrong instant and measured buffer acceptance, not delivery.

Hypothesis, offered by the same turn: that the remaining gap to the ssh time was the transport's. It was not yet supported by anything.

Fact: the retry, timed end-to-end, gave 2.87 s. Fact: that run was "captured before the pkill killed the loop", so the harness's own cleanup fired inside the measurement window.

Fact: the final run gave 0.76 s, hash verified.

So the causal chain runs: a clock placed at the wrong boundary, corrected to the true boundary, then corrected again to a harness that did not shoot its own receiver. The transport was never the variable being observed. The instrument was.

What the tunnel's stats would have shown

The note records no `--stats` capture. What the tunnel *would* have printed is reconstruction, drawn from the crate's own documented field set, and it should be read as such.

A stats line is one logfmt record on stderr, `key=value` pairs carrying their own units, emitted to the process's own stream at a 10 s default cadence (Stats 2026). The fields that matter here are `tx`, `rx`, `tx_pkts`, `rx_pkts`, `sent`, `lost`, `loss`, `cwnd`, `rtt`, `bw_tx`, `bw_rx`, `streams_active`, `streams_total`, `blackholes`, and the role and version identity at the front of the line.

At the sending end a genuinely stalled transfer looks like bytes leaving and nothing coming back, with `cwnd` pinned and `lost` climbing. At the receiving end the same stall looks like `rx` frozen while the writer's file stops growing. The two lines disagree only if you are reading both, which is the argument for the line existing at all: it needs no collector, no socket, no config, just a terminal or journald (Stats 2026).

Two of its properties bear directly on this incident. The counters are per-connection and aggregate. `--stats` and the Prometheus endpoint report the connection's totals and never a per-forward breakdown, so with several `-l` specs up, which stream is the busy one is not a question either surface answers (Limitations 2026). The closing line is conditional. A clean exit prints `final=1`; a `SIGKILL` prints nothing, and absence is a weak signal (Stats 2026). The pkill in this harness was exactly the case that suppresses the record you would most want afterwards.

What the incident does not prove

Nineteen megabytes per second is about 152 Mbit/s. On the emulated 500 Mbit/s, 90 ms path the crate measures a plateau of 470-500 Mbit/s and attributes it to the congestion controller, not the socket buffer; raising `net.core.rmem_default` from its 208 KiB stock to 4 MiB moved nothing (Tuning 2026). At 152 Mbit/s the drop sits at roughly a third of that plateau, and a 14.5 MB transfer is far too short to reach steady state on a long-fat path (Tuning 2026). The winning number is a lower bound on the path, not a verdict on it, and it owes no sysctl.

The buffer ceiling is real but rate-dependent. The same stock buffers that held a 5.6 MB bandwidth-delay product at 500 Mbit/s produced `RcvbufErrors` on unshaped runs at 5.2-7.6 Gbit/s, and the raised windows' bursts found a ceiling the stock 1.25 MB stream window never touched (Tuning 2026). Size for the rate you actually run.

Why one stall beats an average

Three transports, hash-verified, on one path, is a better experiment than most. It is still an average, and averages are what let 0.04 s survive long enough to be typed. Had the runs been pooled, the false reading would have looked like jitter and the headline would have been wrong in the flattering direction.

A single falsified transfer is worth more than any of them, because it names its own failure. The hash said the bytes were fine. The clock said 40 ms. The disagreement between those two statements is the whole diagnosis, and it arrived without a single extra run. The honest sequencing is the one the transcript actually followed: prove carriage, then prove delivery, then time delivery, and never let a timer stop somewhere the file has not reached.

---

12 · ERGONOMICS

Ergonomics: what `ssh -L` gives you, and what parity costs

The claim, and how big it is

qqt's manual claims *the semantics of* `ssh -L` *and* `ssh -R`. The house ergonomics study took that literally, tabulating 56 behaviours across eight areas against `ssh -G`, a live `sshd`, the OpenSSH source, and `./target/release/qqt` (qqt-ssh-ergonomics, §Reading the table).

qqt's whole surface is seven subcommands, three global flags (`-4`, `-6`, `-v`), and four forwarding-relevant flags. The claim holds where a forward is a forward; it does not reach the eighty other behaviours ssh carries, and that edge is where a migrating user's fingers fail.

The door model

A client mode is a *local door* onto one invariant far side: a mutually-authenticated QUIC connection carrying one stream per flow, whose header names the dial target. A mode changes only what the local box presents to its own kernel (qqt-doors, §the mode table). The forward door needs nothing: no `CAP_NET_ADMIN`, no `/dev/net/tun`, nothing an unprivileged rental container can refuse.

Forwarding is the shipped default and stays a contract, identical with every other door present and reachable without a new flag.

The stream contract explains the cheap parity. A flow's first byte is a frame type; a data frame carries `host` and `port` and the receiving side dials them, precisely what a forward needs. A cli-

ent-opened control frame is a `-R` registration the server answers with bind status (qqt-doors, §stream contract).

Declaring a forward

One grammar, four colon-separated fields plus brackets: `<bind>:<port>:<target-host>:<target-port>`, shared verbatim between `-l` and `-R`. `-R` is the same spec with the sides swapped, the mirror ssh gives its four `-L` forms. Unix socket forwarding has no equivalent: the header carries `(host, port)`, and a socket path is neither. `-D` has no flag; a SOCKS5 door sits unwired in `src/socks.rs`.

Three spellings bite. `-L` is not a flag: `error: unexpected argument '-L' found`. A bare port is not a forward: `invalid forward spec '8080'`. And the empty bind inverts: `-l :18099:host:80` resolves to the loopback of the requested family, `127.0.0.1` or `::1`, which is ssh's *omitted*-bind behaviour applied to ssh's *empty* token, so the identical command line binds narrower in qqt (row 1.6; `rationale_qqt-address-family.md`).

A named bind is narrower still: `-l localhost:18060:127.0.0.1:22` resolved to one listener where ssh binds both loopbacks, so ssh's default is reachable in qqt by no spelling (row 1.5). The bracket rule is copied exactly; `-R` honours its literal, where ssh narrows `-R *` to the far loopback under `GatewayPorts no` (row 1.7).

Tearing one down

The listener's lifetime is the connection's. Listeners bind after the QUIC handshake, measured at 100 ms with no listener and 200 ms listening: `forwarding 127.0.0.1:18020 -> 127.0.0.1:22` (qqt-ssh-ergonomics, §8). "Up" means listeners bound, not a master process answering a check.

A `-l` bind failure is fatal: `binding 127.0.0.1:18032: Address already in use`, exit 1. ssh, without `ExitOnForwardFailure`, warns and keeps running; qqt made that option its default and offers no opt-out (row 6.2).

The `-R` half is deliberately asymmetric: a bind the server cannot make is a warning and the `-l` forwards keep running, after a retry of up to about 70 s (qqt-howto, §4). A clean exit drops the server-side listener at once; a client that died without a close leaves the port held until the 60 s idle timeout.

There is no control client, so no `-O cancel` and no `~C` with `-KL`: a forward cannot be added to or removed from a running client. The equivalent of ssh's tilde escape is "-O-shaped, not tilde-shaped" (row 6.5).

Several forwards, several clients

`-l` repeats, one flag per forward; the howto's client carries two `-l` and one `-r` in one invocation (qqt-howto, §4). Duplicate `-R` binds inside one client are refused at startup: `duplicate -R bind 127.0.0.1:18002 — one listener per bind address and port`. Across clients, `-l` listeners are local to the machine, so two hosts may share a local port; `-R` binds land on the server, one per connection, so a server-side port must be unique, and the loser warns `refused by the server` while its `-l` forwards keep running (qqt-howto, §4a).

There is no multiplexing: one connection per client process, no `ControlMaster`, no `ControlPath` tokens. ssh multiplexes because a login costs seconds; a QUIC handshake costs milliseconds. The four-engineer bug that `%r@%h:%p` invites cannot occur (rows 5.1, 5.3).

What the consumer sees

Everything downstream is a plain local TCP listener. That carries most of the ergonomics, and the howto leans on it: keep `ssh -L` armed as a hot spare, because "the two transports are interchangeable from the consumer's point of view: both are plain local TCP listeners" (qqt-howto, §7).

So `scp` and `rsync` need no qqt awareness: they keep their own grammar and port flag, and the port belongs to the forward. qqt's own remote install sidesteps them, the payload arriving on the ssh channel's stdin and running from an `mktemp -d` directory at mode 0700, so no `scp`, `sftp`, or `rsync` subsystem is required at either end (SB-9).

Agent forwarding is out of scope: `ForwardAgent` has no equivalent, the certificate is the credential, and forwarding an agent to a box that runs no ssh would be a second trust path (row 7.10). The `up` path is the exception, its ssh lane passing `IdentitiesOnly=yes` and `BatchMode=yes`, so an agent-held key is invisible unless it is a default `id_*` or `--ssh-key` names it (row 4.2).

Interactive full-screen programs keep their pty, which belongs to the far-side process, not the tunnel. An interactive session is `ssh` through the forward, and `~.` then belongs to that inner ssh. What is lost is a rescue hatch for the tunnel itself: qqt has no pty to inline an escape into, so a wedged client is abandoned with a signal (row 6.5).

The ledger

The gap no row captures, and the one a migrating user feels first, is that qqt has no `-G`. Everything else is learnable from a manual page; "why did `-r mybox` dial that address" is not, and the answer already sits inside the process, because `-r` asks `ssh -G` for it. Related: no `-o`, no config file to override, no `-F`, `-J`, `-W`, or `-O`. A trailing remote command is a stray argument, and the study insists that error be loud. Exit codes are 2 for usage and 1 for a runtime failure, never 255, so a supervisor's `rc == 255` test never fires (row 6.1).

Counted out, the 56 rows land as: copy where a forward is a forward; diverge and document for the empty bind, `up`'s `ssh -F /dev/null` asymmetry, the absent multiplexing, and the exit-code contract; refuse for `-4 -6` together, a trailing command, and a runtime forward-add that would no-op; out of scope for the session layer. The parity is real, and its price is in the table that grants it.

---

13 · THE SECURITY MODEL

The security model: mutual authentication, pinning, and reminting

The trust boundary of a qqt tunnel is a file. There is no account database, no token endpoint, no dial allow-list. A peer is admitted if it presents a chain ending at a CA in the server's trust bundle, and is then trusted as fully as the server's own process. Everything else follows from that (QQT-004).

What the handshake checks

The tunnel runs QUIC v1 over UDP with TLS 1.3 and mutual authentication (RFC 9000). The server requires a client certificate signed by a CA in its bundle; there is no anonymous mode, no plaintext mode, no token. Both ends require the ALPN `quic-tun`, so a mismatch fails the handshake rather than degrading.

The verifier is rustls 0.23 / webpki 0.103 through quinn 0.11. It checks the end-entity validity window at every handshake, on both sides, and rejects a CA certificate presented as a client certificate, so the public CA cert — which ships everywhere — grants nothing without its key. Chains are limited to six subordinate CAs; only the operator's CA key can mint an intermediate.

QUIC's anti-amplification limit allows at most three times the received bytes toward an unvalidated address, so a spoofed-source handshake cannot complete. The server issues no stateless retry, the document's first hardening knob.

Pinning, and what is not pinned

There are two pins, and neither is a key fingerprint. The client pins SNI to `quic-tun` and verifies the server certificate's SAN against that name; the server never inspects the ClientHello's SNI. The direction is explicit: the pin is a client-side mis-issuance tripwire, not a server-side gate, so a certificate minted for anything else is refused (QQT-001). The second pin is the ALPN.

qqt does not record a server key and refuse a change; there is no `known_hosts` equivalent. Verification is signature-and-window against the server-identity CA on every connect, so a server certificate re-minted under the same CA is indistinguishable from the one it replaced.

Who may mint, who may admit

The design splits two powers that are easy to conflate. `gencert` mints a private CA plus role sets; `mint` produces a customer's set under a CA of their own. In the per-customer model the server-identity CA signs only the server certificate, so its key impersonates the door but admits nobody. Each customer's CA key is that customer's admitting power, minted on their box and never returned. Only the public CA cert travels back, for the server's `ca-cert.pem` bundle.

The bundle is the admission store, and it is public. The CA key is the authority. A stolen operator keyset mints no admitting certificate in that model, and the rule that makes it true is explicit: never put the operator's `gencert` CA cert into the server's bundle. The single-CA model collapses the split — one key signs both roles and is fleet-admitting — and the runbook states that trade.

Rotation: the leaf is the clock

Leaves live 365 days by default (`--valid-days`), CAs 3650, so roughly nine leaf generations remint under one CA with no server-side change. `qqt expiry` is the cron-able gate, exiting 1 on an expired or not-yet-valid certificate.

The handshake enforces the leaf's window, so an expired leaf stops admitting on its own. A CA's dates are not checked by the verifier — webpki treats a trust anchor as trusted whatever its dates — so a CA is retired by removing it from the bundle or revoking it, never by waiting.

The routine re-mint copies the retiring leaf aside, because `mint` overwrites `client-cert.pem` in place, then re-mints under the same CA and revokes the retired cert. An old cert stays valid until its window closes; a decommissioned machine still holds a working one. The deny-list is a flat file of SHA-256 fingerprints on the server, re-read at every handshake, so an entry bites at the next connect with no restart. `revoke` is idempotent and fsyncs. The file must not be attacker-writable: whoever can edit it can un-revoke. The check resolves the leaf's issuer from the server's bundle one hop, so revoking a customer's CA kills every leaf it signed, including later mints. A leaf under their own intermediate survives; offboarding that customer means removing their CA from the bundle.

The check sits after the handshake, so a revoked holder's failed connect reads as a closed connection, and a live connection keeps its streams until it drops or the server restarts. The client has no revocation check of the server's certificate at all.

What a stolen client certificate buys

Almost everything the server can see. It dials whatever a certified client names — loopback, the metadata address, internal ranges — no allow-list. A client may hold 1024 concurrent streams per connection, qqt's ceiling against quinn's default of 100, and nothing caps its connections. A `-R` spec binds any address the server process can bind. The mirror is deliberate: the client checks a reverse header against its registered `-R` targets, so a compromised server gains no dial authority the client did not name.

Listener exposure is narrow by default: an empty bind is the requested family's loopback, not a wildcard, and `[::]` under `-6` sets `IPV6_V6ONLY` rather than trusting the host sysctl (QQT-011).

The compensating controls sit outside the protocol: firewall rules on the server's egress, and distinct CAs per trust boundary — a cert from CA-A never authenticates against a server trusting CA-B.

What the tunnel does not cover

It is not a VPN in the authorization sense. Confidentiality and integrity come from TLS 1.3, but target authorization does not exist. It is not a traffic-analysis shield: the outer QUIC flow's packet sizes and timing stay visible to anything on the path, and the documents claim confidentiality and integrity, not shape-hiding. It is not an audit system — every holder is trusted equally, so the certificate is an admission ticket, not an identity. The operator machine is inside the trust boundary too: anyone with write access to the cert directory can swap PEMs; file modes (keys `0600`, certs `0644`, directories `0700`) and atomic writes are the only guard named.

Against ssh and known_hosts

The idea matches ssh host keys: the client must decide the far end is the machine it meant. The mechanisms differ in where the decision lives. `known_hosts` records a key per host name, trusts it on first use, and shouts when it changes; that trust is local, per-client, and cheap to click through. qqt replaces the record with a CA: the client verifies a chain on every connect, and a change of server key under that CA produces no event at all.

That inversion moves the failure. ssh's failure mode is the warning nobody reads, and the host key that changes because someone rebuilt a box. qqt's is quieter: a leaked server key keeps verifying until the CA rotates, and a mistaken certificate for `quic-tun` is accepted by every client trusting that CA. In exchange, qqt gets fleet admission and revocation in one bundle and one file, where ssh needs a per-host edit or a CA of its own. ssh asks you to notice a change; qqt asks you to guard a CA.

14 · WHERE IT DOES NOT HELP

Where it does not help: the honest limits

The claim this chapter polices

"QUIC is faster than TCP" is false unqualified. QUIC is faster in a regime: loss, long round trips, several concurrent flows. Outside it, it is the same speed or slower, and a tunnel built on it inherits those edges.

Where UDP is somebody else's port

QUIC rides UDP "to better facilitate deployment in existing systems and networks" (RFC 9000 §1), and those systems get a vote. Middleboxes "block or throttle" UDP, so "many real-world deployments observe QUIC connections being silently dropped or degraded" (Sachdeva et al. 2026). Across 2,600 streaming hours of YouTube, "the legacy setup (TCP) either outperforms or performs the same as the QUIC-enabled browser for more than 60% of cases" (Sachdeva et al. 2022). Where QUIC is filtered on purpose it is filtered by name: the Great Firewall censors QUIC on SNI (Zohaib et al. 2025).

qqt has no automatic fallback. A blocked UDP port produces a 10 s connect deadline and an error naming the address, and with `--reconnect` the client retries on a fixed 2 s cadence forever. The escape hatch is a manually armed `ssh -L` hot spare on another local port. The flip is cheap but manual: an operator action, not a protocol feature. No happy-eyeballs race either: a dead first family costs a connect deadline per attempt (rationale_qqt-address-family).

The MTU budget the tunnel spends first

RFC 9000 §14 is a hard floor: "QUIC MUST NOT be used if the network path cannot support a maximum datagram size of at least 1200 bytes," and "UDP datagrams MUST NOT be fragmented at the IP layer." Every encapsulation layer spends that budget first. The usable payload is what is left of the path MTU after each header on the stack, and each one is an exact, countable charge:

$$P_{\text{quic}} = \text{MTU} - H_{\text{ip}} - H_{\text{udp}} - (1 + L_{\text{cid}} + L_{\text{pn}} + 16) \tag{14}$$

The encapsulation budget. Each term is an exact header charge, not an estimate.

Worked for the two stacks a tunnel actually meets on a 1500-byte path, with an 8-byte destination connection ID and a 2-byte packet number:

$$\begin{aligned} \text{IPv4:}&\quad 1500 - 20 - 8 - (1 + 8 + 2 + 16) = 1445\ \text{B} \\ \text{IPv6:}&\quad 1500 - 40 - 8 - (1 + 8 + 2 + 16) = 1425\ \text{B} \end{aligned} \tag{15}$$

MTU arithmetic. Every header on the stack is a charge against the path MTU, and the QUIC short header carries a 1-byte flags field, the destination connection ID, the packet number and a 16-byte AEAD tag (RFC 9000 §17.3.1, RFC 9001 §5.3).

At QUIC's own 1200-byte floor the IPv6 stack leaves 1121 bytes of stream data once the 4-byte STREAM frame is charged (RFC 9000 §19.8), and that is the number an inner TCP connection's MSS must fit under.

On a WireGuard-carried path qqt's own stats line reports `mtu=1389`, where a direct path may carry the full 1452-byte transmitted payload (`MtuDiscoveryConfig::upper_bound`). The tun-

nel's interface door spends more: `--tun-mtu 1280` leaves exactly 1232 bytes of UDP payload, and because the stack has no IPv6 egress fragmentation an oversize v6 datagram is refused with a typed error rather than fragmented.

A tunnel that clamps the path MTU also swallows DPLPMTUD's probes, so the path settles below what it could carry. A path that cannot carry 1200 bytes after encapsulation is not a slow qqt path; it is not a qqt path.

CPU is charged per packet

"QUIC authenticates the entirety of each packet and encrypts as much of each packet as is practical" (RFC 9000 §1); the AEAD tag adds "16 bytes larger than their input" (RFC 9001 §5.3). ACK-only packets pay the tag too, where a TCP ACK is a bare header the kernel emits and forgets. Header bytes are a wash (RFC 9002 §7.2 credits UDP's 8-byte overhead against TCP's 20); packets are not.

WHAT WAS MEASURED	RESULT
`qqt CPU cost`	~6.3 CPU-ms per MB, about 0.8 of a core per Gbit/s
`Userspace qqt throughput`	378–469 Mbit/s
`Kernel datagram tunnel, same harness`	872–1050 Mbit/s

Zhang et al. (2024) name the same cost: "high receiver-side processing overhead, in particular, excessive data packets and QUIC's user-space ACKs." Kernel crypto buys roughly 2.3–2.8× here. On a box near its packet budget, that is the whole story.

The pacing gap, and no datagrams

qqt forwards TCP connections. A datagram-shaped workload — DNS, a voice call, another QUIC session — does not ride the stream door at all; it needs the TUN door, at the cost of `CAP_NET_ADMIN` and `/dev/net/tun`.

Pacing is the second gap. quinn's bbr "paces at a cwnd-derived rate instead of" the bandwidth estimate its design assumes, so nothing ties cwnd to the path; the measured symptom is a sender cwnd that read 687 MB across clean cells. The shipped 64 MiB connection window bounds it, but flow-control credit is not a rate model. Host-side, "plain `fq` does not pace UDP"; the mitigation is `tc fq maxrate <path rate>`, an operator's hand.

One window, shared by every stream

QUIC partitioned delivery, not rate. RFC 9000 §9.4 permits "a single congestion control context and a single loss recovery context," and RFC 9002 §7 fixes it per path. A loss halves that one window (RFC 9002 §7.3.2) for every stream writing into it, and the spec says so plainly: "when data from multiple streams is included in a single QUIC packet, loss of that packet blocks all those streams." On top sits a connection-wide flow-control budget (RFC 9000 §4.1), so a stalled reader throttles its neighbours.

qqt is one connection, therefore one congestion domain and one unit of death: an idle timeout or a network change ends every forwarded stream at once. And the shipped default carries a fairness cost — bbrv1 "takes roughly twice its fair share against cubic co-flows on a shared bottleneck." On a household uplink that is a real neighbour tax.

A tunnel the TCP tools cannot see

There is no per-stream accounting. `--stats` and the Prometheus endpoint report per-connection aggregates; which `-l` spec is busy is a question neither surface answers. The flow you want to inspect exists only as bytes inside one UDP socket, so `ss` shows a listener, `tcpdump` shows ciphertext, and conntrack sees one UDP 5-tuple where you expected five flows. On gVisor-sandboxed containers `/proc/net/udp` may not show the socket and `ss` may be absent; the log's `listening on udp` line is the proof.

A silently dropped path is a black hole for 60 seconds; a clean stop is instant, a middlebox that forgets UDP state is not.

When plain ssh -L is simply better

The production record's 5.0× (47.7 ms qqt p50 against 240.0 ms relayed `ssh -L` p50) is a topology result. The relay multiplies round trips; the tunnel removes them. On a 41.5 ms direct path there is no relay to remove, and the record says so: "a direct-ssh path will not show this gap." On a clean single-flow path the comparison inverts: raw TCP moved 337/336 Mbit/s where qqt moved 100/102. The setup saving is one round trip, once. On a short, clean, single-flow path `ssh -L` is already at the ceiling, and qqt charges CPU to arrive at the same place.

Do not deploy qqt when

CONDITION	WHY
`UDP is blocked, throttled or SNI-filtered on the path`	No automatic fallback; only a hand-flipped `ssh -L` spare
`Path PMTU under 1200 bytes after encapsulation`	RFC 9000 §14 forbids it outright
`One flow on a clean, short path`	3× slower than raw TCP; no relay to remove
`Sustained multi-Gbit bulk on a CPU-tight box`	~0.8 core per Gbit/s, 2.3–2.8× kernel crypto
`Windows endpoints`	Unsupported; unix permission bits are load-bearing
`Cert holders you do not fully trust`	No dial allow-list, no CRL/OCSP, no per-stream caps past 1024
`Per-flow accounting or capture`	Counters are per-connection aggregates
`Multipath or link bundling`	One QUIC path, no multipath
`A UDP port exposed to a hostile internet`	No stateless retry; hardening left to the operator

15 · A SCENARIO MODEL AND A DECISION GUIDE

A scenario model and a decision guide

The door is fixed; the path under it is what varies. A renter reaches the box through the provider's SSH tunnel wherever the box sits, and qqt replaces that tunnel on a long relay, a short route or a lossy uplink without changing what the door does. This chapter is the instrument for that decision: which paths pay for a QUIC carrier and which do not. Every figure is marked (R) for a measurement in the benchmark record or (M) for arithmetic on top of one. The band edges in the decision table are neither: they are editorial, bracketing the measured points.

The scenario, and its arithmetic

The mechanism note fixes the traffic: a 3.0 GB bulk download, a chat stream at 1.0 KB/s, a JSON-RPC call every two seconds, and one reply that thinks for 3.01 s, from a European residential uplink to a US East 4×B200 rental (07-scenario-model.md). The direct round trip is 41.5 ms; the platform's relayed `ssh -L` path is 90.4 ms, a 48.9 ms premium that is 54% of the relayed path (R).

Three of those four flows are 0.1% of the bytes and 100% of the interactivity (M). That census is the decision: the door carries all four, so a carrier that improves only bulk improves the traffic nobody watches, and one that improves only the small flows leaves the transfer unfinished.

The arithmetic predicts every result here. Throughput is window over round trip, so the relayed socket plateaus at 2.0 MB, which is 177 Mbit/s times 90.4 ms (M). When the cold reply stalls at the head of that one byte stream, the queue fills at the excess offered rate, (504 − 177) Mbit/s = 41 MB/s, and clamps full 49 ms into the stall, long before a human notices (M). It drains in one bandwidth-delay product, 90.4 ms. Measured round trip for every flow on that socket goes 90.4 → 180.8 → 271.2 ms (M). The QUIC panel obeys the same law with better constants: a 2.6 MB window, a ~63 KB queue, and an idle latency that rises by the measured +4.3 ms (R).

Two predictions. Interactive gain tracks round trips removed, not window: a 4 KB reply every two seconds leaves the link 97% idle. Bulk gain tracks window and loss, and on a clean path there is nothing to recover.

Sensitivity

Round trip sets both terms: the interactive gain directly, the bulk ceiling inversely. Decomposed, the production gain is 2.65 relayed round trips per request at a 48.9 ms premium, about 130 ms, plus 1.5 extra round trips at 41.5 ms, about 62 ms (R). On a direct-port cloud the 130 ms term disappears and the 62 ms term remains (M): that is the difference between the headline 5.0× and a direct-path gain nearer 2×.

Loss makes the controller the dominant variable. At 1% loss with eight streams, cubic delivers 2.5–8.6 Mbit/s and bbr 336–367 on the same shape, a 40–130× gap from the controller alone (R). `ssh -L` delivers 0.0–3.7 Mbit/s at every loss profile with eight streams (R). Below loss the ordering inverts: on a clean path a single stream through qqt moves 100/102 Mbit/s against raw TCP's 337/336 and `ssh -L`'s 169/150 (R).

The mix decides which number the user experiences, and forces no trade: under a bulk upload, 32 small streams held a p99 inter-chunk gap of 11.0 ms inside the window against 12.5 ms outside (R).

One regime is unresolved: in the ±10 ms jitter profile qqt's inner streams sent 23 Mbit/s and delivered under 1 (R), from a single run the record calls measured once, mechanism uncharacterized.

The decision table

The latency band is the round trip of the path you would otherwise run, relayed or not, because that is the path the gain is measured against.

PATH RTT × LOSS	EXPECTED GAIN VS `ssh -L`	`--cc`	TUNING THAT MATTERS
≤40 ms, clean	interactive ~1× — `ssh -L` over a 0.1 ms path costs 47.6 ms against qqt's 47.7 (R); bulk ~1×	bbr	none; 4 MiB is past the BDP
≤40 ms, 0.1–1%	interactive ~1×; bulk 5–7× single stream, 28.9 vs 4.1 Mbit/s (R)	bbr	`--cc`, not window, decides it
≤40 ms, >1%	bulk 1.5–3×, 6.8 vs 2.3 at 1% (R); absolute rates single-digit	bbr	`tc fq maxrate`; expect little
40–100 ms, clean	interactive 2–5× (2× modelled, 5.0× relayed measured); bulk 1.5–1.8× at 8 streams, 2.85× relayed (R)	bbr	default window; buffers only above ~5 Gbit/s
40–100 ms, 0.1–1%	bulk 2.7× down, `ssh -L` upload dead (R)	bbr	cubic only beside cubic co-flows
40–100 ms, >1%	bulk 40–130×, 336–367 against 0.0–3.7 (R)	bbr	`fq maxrate`; run lossy cells 20 s
>100 ms, clean	interactive 2.3× on probes, 4.7× on keystrokes (R)	bbr	4 MiB window is the measured sweet spot
>100 ms, 0.1–1%	`ssh -L` 0.0–3.7; qqt 322–336 at 90 ms (R); longer RTT untested (M)	bbr	smaller window on a ≤100 Mbit/s uplink
>100 ms, >1%	same regime; a single stream holds 1.3–6.8 Mbit/s at 1–5% (R)	bbr	accept single-digit rates; `fq maxrate`

The tuning that matters

The window. A single stream cannot exceed window over round trip. At the stock 1.25 MB window a stream stops near 111 Mbit/s on a 90 ms path (measured 99–104); the shipped 4 MiB lifts it to ~324 (R). Eight MiB breaks the cubic aggregate, 57.7 where 4 MiB delivers 453, because the receiving kernel drops the burst (R).

The controller. bbr is the right default: the lossy cells are why. The exception is load-bearing for anyone on a home uplink: bbrv1 takes roughly twice its fair share against cubic co-flows on a shared bottleneck, so on a shared uplink with household cubic co-flows, run **`--cc cubic`** with eyes open (rationale_qqt-congestion-control.md). Cubic is well-mannered when it is not the only flow on a lossy long path.

Kernel socket buffers. Measure before touching them. With the path capped at 500 Mbit/s and a 5.6 MB BDP, stock 208 KiB defaults delivered 471–475 Mbit/s at every setting tested (R). The ceiling is rate-dependent: the same pair unshaped at 5.2–7.6 Gbit/s moved `RcvbufErrors` again (R). Size for the rate you run; `*_max` alone changes nothing for qqt.

The host qdisc. `tc fq maxrate <path rate>` caps the UDP flow's bursts on the datacenter side, the one item from ESnet's long-fat guidance that transfers to a userspace QUIC tunnel (instructions_qqt-tuning.md).

What no flag fixes

Three deficits are structural. On a clean short path qqt is slower for single-stream bulk, 100 against 337 Mbit/s (R). A UDP-blocking network gets no gain, only a fallback. And a gap that came from a relay is removed only by removing the relay; the 5.0× is a topology result.

Procedure

Measure the round trip and loss of the path you would run. Clean, short, single-stream: do not deploy. Otherwise deploy with defaults (bbr, 4 MiB), cap the qdisc, and size the window to the BDP.

---

16 · REPRODUCTION, OPEN QUESTIONS, AND THE BIBLIOGRAPHY

Reproduction, open questions, and the bibliography

What a minimal reproduction needs

Three things, and no fourth. Two boxes that can reach each other, one UDP port open end to end, and a path you can make lossy on demand.

The first box runs `qqt server`; the second runs `qqt client -l <bind>:<port>:<target-host>:<target-port> -r <server>:<port>`. The `-l` spec is four fields, not two — a bare `host:port` is rejected (`rationale_qqt-benchmarks.md`). Certificates come from `qqt gencert -o <dir> -f`, split into a server half and a client half.

The far end must be reached two ways: over the tunnel's UDP port, and over whatever you are comparing against — the provider's SSH relay on the keystroke work, direct `ssh` to the public address on the AWS pair. The single UDP port in the security group is what makes that pairing honest: both panes are the same `sshd` on the same box, so the shell, the key and the far end cancel and only the way there differs (`qqt-vs-raw-ssh-keystroke-latency/index.md`).

A lossy link, when no real one is available, is a pair of network namespaces joined by a veth pair with `tc netem` shaping both directions — the shipped harness uses 90 ms RTT with configurable loss and jitter, and `htb` for rate (`examples/bench-loss-matrix.sh`). It cannot reproduce the relay tax, because a local `ssh -L` is a direct connection; it can reproduce the loss regimes where the transports separate.

The AWS side-by-side rig

Four scripts drive the filmed pair, and all four take their ssh commands and ports from the environment. `rig-qqt-aws.sh` builds the screen: Xvfb at 1920×1080, one alacritty at 190×38 and 12.5 pt, a tmux session named `qqt-rig` with two panes and `synchronize-panes` on, a `screenkey` band 1920×220 at y=860 with a 0.45 s hold, and `rig-overlay.py` drawing an 84-pixel header that names each transport and carries a clock drawn into the framebuffer itself.

Two layouts, one per scene family. `up-remote` puts both panes on the box for typing and `vi` — the lag you can see. `up-local` puts both panes here, each holding its own `-L` forward to its own iperf3 server on the far end, for throughput.

Two traps are written into the script's own comments. An iperf3 server runs one test at a time, so the two paths get two server instances; sharing one would serialize the panes and destroy the comparison. And the forwards run with `ControlMaster=no` and `ControlPath=none`, because a shared mux socket keeps a `-L` bound to its local port after the launcher exits — the next layout then opens on a port already held by a dead forward's tunnel and both panes point at one backend.

`rig-qqt-aws-ab.sh` runs the interleaved latency blocks through `examples/interactive_probe.py`: 60 samples per transport, blocks of ten, alternating which transport goes first so a drifting path moves both columns together. `rig-qqt-aws-record.sh` records one scene.

The echo-server harness

The tunnel benchmark pushed 200 KB up to a byte-echo server and back down, once per second, for five minutes — 300 bursts, zero failures, 61.44 MB each way. The server is `python3 echo_server.py 19001`, byte-for-byte with no framing; both ends ran `--stats 5`. Two tunnels were measured, deliberately not nested inside each other, over `10.10.0.57:18443` (QUIC inside WireGuard) and `37.27.134.57:18445` (QUIC direct to a public IP).

The 63-byte MTU difference — 1389 on the nested path, 1452 on the public one — is the proof the paths are genuinely different. Had both reported the same MTU, the "separate paths" premise would have been wrong. The Linux binary was shipped from a Linux checkout; the macOS build does not run on Linux (`qqt-tunnel-benchmark/index.md`).

The 60 fps capture, and what it can resolve

Capture is `ffmpeg -f x11grab -video_size 1920x1080 -framerate 60 -c:v libx264 -preset ultrafast -qp 0` — lossless, so frame boundaries survive the encoder. Frame quantization is 16.7 ms.

Reading the keypress overlay's pulse against each pane's glyph rise off the frames gives qqt 50 ms p50 and 50 ms p90, against relayed ssh 217 ms p50 and 283 ms p90. Because the clock is in the framebuffer, any frozen frame's timestamp is that frame's own. The honest reading of the left column is "one to three frames" and of the right "a dozen or more". The clip and the CSV agree because they measure the same event; the clip is a second measurement, not a decoration on the table.

What each experiment left behind

EXPERIMENT	ARTIFACTS
keystroke latency	`summary.tsv` (n, p50, p90, p99, min, max, mean per cell and transport), `samples.csv.gz` (1,440 individual samples), five MP4 clips, `latency-chart.png`, four rig scripts
tunnel benchmark	four stats blocks quoted verbatim in the note, plus a numbered reproduce block
emulated matrix	one `summary.tsv` row per cell (profile × transport × streams × direction), per-run JSON in `~/qqt-bench/results/`, CPU ticks for qqt runs, `matrix.log`
concurrency	`examples/bench-concurrency.py`, `serve` and `run --mode m1|m2|m3|m4`
production path	no script; the note gives the `curl` and `iperf3` invocations, 20+ runs per transport, interleaved

Five open questions

1. Is the nested path's win a path effect or a time effect? The WG and public runs were sequential — 15:05–15:10 and 15:10–15:15 UTC — not simultaneous. Some of the 285 µs against 37–49 ms jitter gap could be time-varying congestion. A simultaneous A/B removes the confound and is quick to do.

2. What is happening in the jitter profile? Under 1 % loss with ±10 ms jitter, qqt's inner streams sent 23 Mbit/s and delivered under 1. Heavy netem reordering starved delivery. Single 10 s runs with possible distribution artifacts: measured once, mechanism uncharacterized.

3. What does the UDM's routed path do to this? The same tunnel test through the gateway rather than over the mesh is the number that decides whether the router-VM design pays for itself. No such run exists in this volume.

4. Is there a real ceiling? The tunnel benchmark was a quality test: 200 KB/s offered, 1.6 Mbit/s asked for, not what the tunnel can do. Finding the ceiling needs a saturated blast.

5. Does macOS behave? It was not part of the emulated matrix — no UDP GSO, so quinn falls back — and no throughput delta is claimed there.

— · 43 SOURCES

References

ALPHABETICAL · CHICAGO · HANGING INDENT · RETRIEVAL DATES ON UNDATED WEB SOURCES

[1] Allman, Mark, Vern Paxson, and Ethan Blanton. 2009. "TCP Congestion Control." RFC 5681. https://www.rfc-editor.org/rfc/rfc5681.html.

[2] Belshe, Mike, Roberto Peon, and Martin Thomson, eds. 2015. "Hypertext Transfer Protocol Version 2 (HTTP/2)." RFC 7540. https://www.rfc-editor.org/rfc/rfc7540.html.

[3] Bishop, Mike, ed. 2022. "HTTP/3." RFC 9114. https://www.rfc-editor.org/rfc/rfc9114.html.

[4] Cardwell, Neal, Yuchung Cheng, C. Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. 2016. "BBR: Congestion-Based Congestion Control." ACM Queue 14 (5): 20–53. https://doi.org/10.1145/3012426.3022184.

[5] Cheng, Yuchung, Neal Cardwell, Soheil Hassas Yeganeh, and Van Jacobson. 2024. "BBR Congestion Control." Internet-Draft draft-ietf-ccwg-bbr-00. IETF, July 19, 2024. https://datatracker.ietf.org/doc/draft-ietf-ccwg-bbr/.

[6] Eddy, Wesley, ed. 2022. "Transmission Control Protocol (TCP)." RFC 9293. https://www.rfc-editor.org/rfc/rfc9293.html.

[7] Edeline, Korian, Mirja Kühlewind, and Benoit Donnet. 2016. "Measuring and Deploying QUIC." ETH Zurich TIK Technical Report 366, December 2016. https://arxiv.org/abs/1612.07816.

[8] Eggert, Lars, Godred Fairhurst, and Greg Shepherd. 2017. "UDP Usage Guidelines." RFC 8085. https://www.rfc-editor.org/rfc/rfc8085.html.

[9] Ford, Alan, Costin Raiciu, Mark Handley, Olivier Bonaventure, and Christoph Paasch. 2020. "TCP Extensions for Multipath Operation with Multiple Addresses." RFC 8684. https://www.rfc-editor.org/rfc/rfc8684.html.

[10] Gettys, Jim, and Kathleen Nichols. 2012. "Bufferbloat: Dark Buffers in the Internet." Communications of the ACM 55 (1): 57–65. https://doi.org/10.1145/2063176.2063196.

[11] Goel, Utkarsh, Moritz Steiner, Mike P. Wittie, Stephen Ludin, and Martin Flack. 2017. "Domain-Sharding for Faster HTTP/2 in Lossy Cellular Networks." arXiv:1707.05836, July 18, 2017. https://doi.org/10.48550/arXiv.1707.05836.

[12] Harkanson, Ryan, Yoohwan Kim, Ju-Yeon Jo, and Christopher Pham. 2019. "Effects of TCP Transfer Buffers and Congestion Avoidance Algorithms on the End-to-End Throughput of TCP-over-TCP Tunnels." In Advances in Intelligent Systems and Computing. Springer. https://doi.org/10.1007/978-3-030-14070-0_55.

[13] Honda, Osamu, Hiroyuki Ohsaki, Makoto Imase, Mika Ishizuka, and Junichi Murayama. 2005. "Understanding TCP over TCP: Effects of TCP Tunneling on End-to-End Throughput and Latency." In Proceedings of SPIE 6011, 60110H. https://doi.org/10.1117/12.630496.

[14] Hätönen, Seppo, Aki Nyrhinen, Lars Eggert, Stephen Strowes, Pasi Sarolahti, and Markku Kojo. 2010. "An Experimental Study of Home Gateway Characteristics." In Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement (IMC '10), 260–66. https://doi.org/10.1145/1879141.1879174.

[15] Iyengar, Jana, and Ian Swett, eds. 2021. "QUIC Loss Detection and Congestion Control." RFC 9002. https://www.rfc-editor.org/rfc/rfc9002.html.

[16] Iyengar, Jana, and Martin Thomson, eds. 2021. "QUIC: A UDP-Based Multiplexed and Secure Transport." RFC 9000. https://www.rfc-editor.org/rfc/rfc9000.html.

[17] Kakhki, Arash Molavi, Samuel Jero, David Choffnes, Cristina Nita-Rotaru, and Alan Mislove. 2017. "Taking a Long Look at QUIC: An Approach for Rigorous Evaluation of Rapidly Evolving Transport Protocols." In Proceedings of the Internet Measurement Conference (IMC '17), 290–303. https://doi.org/10.1145/3131365.3131368.

[18] Langley, Adam, Alistair Riddoch, Alyssa Wilk, Antonio Vicente, Charles Krasic, Dan Zhang, Fan Yang, et al. 2017. "The QUIC Transport Protocol: Design and Internet-Scale Deployment." In Proceedings of SIGCOMM '17, 183–96. https://doi.org/10.1145/3098822.3098842.

[19] Marx, Robin. 2020. "Head-of-Line Blocking in QUIC and HTTP/3: The Details." Web Performance Calendar, December 2020. https://calendar.perfplanet.com/2020/head-of-line-blocking-in-quic-and-http-3-the-details/.

[20] Naylor, David, Alessandro Finamore, Ilias Leontiadis, Yan Grunenberger, Marco Mellia, Maurizio Munafò, Konstantina Papagiannaki, and Peter Steenkiste. 2014. "The Cost of the 'S' in HTTPS." In Proceedings of the 10th ACM International Conference on Emerging Networking Experiments and Technologies (CoNEXT '14), 133–40. https://doi.org/10.1145/2674005.2674991.

[21] Nichols, Kathleen, and Van Jacobson. 2012. "Controlling Queue Delay." Communications of the ACM 55 (7): 42–50. https://doi.org/10.1145/2209249.2209264.

[22] Nichols, Kathleen, Van Jacobson, Andrew McGregor, and Jana Iyengar. 2017. "Controlled Delay Active Queue Management." RFC 8289. https://www.rfc-editor.org/rfc/rfc8289.html.

[23] Oku, Kazuho, and Lucas Pardue. 2022. "Extensible Prioritization Scheme for HTTP." RFC 9218. https://www.rfc-editor.org/rfc/rfc9218.html.

[24] Pauly, Tommy, Eric Kinnear, and David Schinazi. 2022. "Unreliable Datagram Extension to QUIC." RFC 9221. https://www.rfc-editor.org/rfc/rfc9221.html.

[25] Paxson, Vern, Mark Allman, H. K. Jerry Chu, and Matt Sargent. 2011. "Computing TCP's Retransmission Timer." RFC 6298. https://www.rfc-editor.org/rfc/rfc6298.html.

[26] Postel, Jon, ed. 1980. "User Datagram Protocol." RFC 768. https://www.rfc-editor.org/rfc/rfc768.html.

[27] Quickquic. 2026. "The Benchmark Record." docs/rationale_qqt-benchmarks.md, quickquic crate. /home/dev/ventures/quickquic/docs/rationale_qqt-benchmarks.md.

[28] Quickquic. 2026. "qqt Limitations: The Honest List." docs/architecture_qqt-limitations.md, quickquic crate. /home/dev/ventures/quickquic/docs/architecture_qqt-limitations.md.

[29] Quickquic. 2026. "qqt Tuning: UDP Buffers, the BDP, and What Is Worth Setting." docs/instructions_qqt-tuning.md, quickquic crate. /home/dev/ventures/quickquic/docs/instructions_qqt-tuning.md.

[30] Quickquic. 2026. "qqt Wire Protocol v2." docs/architecture_qqt-protocol.md, quickquic crate. /home/dev/ventures/quickquic/docs/architecture_qqt-protocol.md.

[31] Quickquic. 2026. "The Stats Line." docs/rationale_qqt-stats.md, quickquic crate. /home/dev/ventures/quickquic/docs/rationale_qqt-stats.md.

[32] Rüth, Jan, and Oliver Hohlfeld. 2019. "A Performance Perspective on Web Optimized Protocol Stacks: TCP+TLS+HTTP/2 vs. QUIC." In Proceedings of the Applied Networking Research Workshop (ANRW '19), 1–7. https://doi.org/10.1145/3340301.3341123.

[33] Sachdeva, Mayank, Anirudh Mondal, and Sandip Chakraborty. 2022. "YouTube over Google's QUIC vs Internet Middleboxes: A Tug of War between Protocol Design and Network Operations." arXiv:2203.11977. https://doi.org/10.48550/arXiv.2203.11977.

[34] Seufert, Michael, Ralf Schatz, Nikolas Wehner, and Pedro Casas. 2019. "QUIC and TCP: A Performance Benchmark." In Proceedings of the 2019 IEEE Conference on Network Softwarization (ICIN), 1–6. https://doi.org/10.1109/ICIN.2019.8685913.

[35] Stewart, Randall, ed. 2007. "Stream Control Transmission Protocol." RFC 4960. https://www.rfc-editor.org/rfc/rfc4960.html.

[36] Thomson, Martin, and Sean Turner. 2021. "Using TLS to Secure QUIC." RFC 9001. https://www.rfc-editor.org/rfc/rfc9001.html.

[37] Titz, Olaf. 2001. "Why TCP Over TCP Is a Bad Idea." Web essay, last modified April 23, 2001. Archived at https://web.archive.org/web/20181226095658/http://sites.inka.de/sites/bigred/devel/tcp-tcp.html.

[38] Trevisan, Martino, Danilo Giordano, Idilio Drago, and Ali Safari Khatouni. 2021. "Measuring HTTP/3: Adoption and Performance." In Proceedings of the 2021 Mediterranean Communication and Computer Networking Conference (MedComNet), 1–8. https://doi.org/10.1109/MedComNet52149.2021.9501274.

[39] Wolsing, Konrad, Jan Rüth, Klaus Wehrle, and Oliver Hohlfeld. 2019. "A Performance Perspective on Web Optimized Protocol Stacks: TCP+TLS+HTTP/2 vs. QUIC." In Proceedings of the Applied Networking Research Workshop (ANRW '19), 1–7. https://doi.org/10.1145/3340301.3341123.

[40] Wu, Yuchung, Hwanju Song, and Yee-Ting Li. 2023. "CUBIC for Fast and Long-Distance Networks." RFC 9438. https://www.rfc-editor.org/rfc/rfc9438.html.

[41] Xu, Lixia, Keith Winstein, and Balakrishnan Chandrasekaran. 2022. "QUIC Congestion Control." RFC 9308. https://www.rfc-editor.org/rfc/rfc9308.html.

[42] Zhang, Xumiao, Shuowei Jin, Yi He, Ahmad Hassan, Z. Morley Mao, Feng Qian, and Zhi-Li Zhang. 2024. "QUIC Is Not Quick Enough over Fast Internet." In Proceedings of the ACM Web Conference (WWW '24), 2713–22. https://doi.org/10.1145/3589334.3645323.

[43] Zohaib, Muhammad, et al. 2025. "Exposing and Circumventing SNI-Based QUIC Censorship of the Great Firewall of China." In Proceedings of USENIX Security '25. https://www.usenix.org/conference/usenixsecurity25.

This volume collects the house's work on quickquic, the Quick QUIC Tunnel (qqt): a TCP forwarder that carries each forwarded connection as one stream on a mutually authenticated QUIC connection. It is set from sixteen chapters written against the crate's own documents and the research notes the measurements left behind, and it is published on the One Frontier Foundation research shelf at onefrontier.org/research/quickquic-research.pdf. Numbers are quoted as the sources state them. Where a figure is a model rather than a measurement, the chapter says so. Where the evidence is thin, the chapter says that too. Original research from onefrontier.org. Text, data and figures in this volume were assembled from the qqt crate, its benchmark corpus and the published literature, and are published here first. Published November 20, 2026. Research current through November 20, 2026. ONEFRONTIER · VOL. 1, NO. 1 · NOVEMBER 2026.