QUEST UNIVERSITY CANADA

Department of Physics, Mathematics and Computer Science

PROTOCOL SPECIFICATION · v2

BOLT

Buffered Optimized Link Transport

A UDP-Based Bulk File Transfer Protocol with Application-Aware Rate Control

D. M. Leatti

4 November 2012

DOI: 10.5281/zenodo.20351767

PER ASPERA AD SAPIENTIAM

Abstract

BOLT is a UDP-based file transfer protocol designed for single-flow bulk transfer between authenticated peers over high-latency, lossy links. The motivating problem is the well-known underperformance of single-flow TCP under exogenous packet loss: on a 50 Mbps, 227 ms, 11%-loss satellite link, an 89 GB transfer over single-flow TCP requires more than forty days; BOLT completes the same transfer in 4.5 hours. The contributions are a state-machine rate controller ("Hose") with explicit memory of the last loss cliff, eliminating a sawtooth pathology that loss-blind probing produces under exogenous loss; per-packet authenticated encryption with derived non-transmitted nonces, saving twelve bytes per encrypted packet versus DTLS-style record protection; a nine-byte data header; lock-free bitmap-based deduplication and resume; and platform-aware send paths exploiting Linux UDP segmentation offload where available and a Windows spin pacer where it is not. The protocol is evaluated against three TCP regimes (classic, modern with BBR and RACK-TLP, and PEP-accelerated) on the operating conditions of a deployed Patagonian meteorological station and recovers throughput comparable to PEP-accelerated TCP without requiring operator infrastructure or terminating end-to-end semantics. The reference implementation is in production use.

1. Introduction

Eighty-nine gigabytes of meteorological radar archive moves from a station in Patagonia to a data center in Buenos Aires across a satellite uplink of approximately 50 Mbps capacity, 227 ms round-trip time, and 11% independent packet loss. A single TCP flow over this link, modeled per the throughput approximation of Mathis et al. [1], sustains 0.189 Mbps and completes the transfer in 43.5 days. The same file over BOLT completes in 4.5 hours.

The gap is two to three orders of magnitude, and it is not a quirk of any one stack. It is the predictable consequence of a transport protocol designed in 1988 for a different problem [2], whose later loss-recovery mechanisms (CUBIC [3], BBR [4], RACK-TLP [5]) modernize the response to loss without questioning the underlying assumption that loss reliably signals contention. On a link where loss is exogenous (atmospheric attenuation on a satellite hop, microwave fading, radio bit errors that the physical layer cannot mask), the assumption is wrong, and every mechanism built on it overreacts.

The conventional operator response is the performance-enhancing proxy: a middlebox terminates the TCP connection mid-path, generates local acknowledgments toward the sender, and runs a tuned transport over the bad hop [6]. This works (an estimated 39 Mbps in the Patagonia scenario), but it requires a deployed middlebox the user must trust, it terminates end-to-end TCP semantics, and it defeats end-to-end encryption. For a sensor network owned by the data destination, deploying middleboxes on every satellite gateway in the country is not a serious option.

This paper describes BOLT (Buffered Optimized Link Transport), a UDP-based file-transfer protocol designed for single-flow bulk transfer between authenticated peers over hostile links. BOLT moves the file at 43.7 Mbps in the Patagonia scenario, within 12% of the throughput a PEP-accelerated TCP achieves, while preserving end-to-end semantics, end-to-end encryption, and a single-binary deployment model that requires no operator infrastructure.

The contributions are:

1. A purpose-built file-transfer protocol with a 9-byte DATA header, a fixed 1400-byte MTU, and 1-RTT session establishment.
2. The Hose rate controller, a state-machine rate controller with explicit memory of the last loss cliff. The cliff-memory design eliminates a pathological sawtooth that loss-blind probing produces on links with non-zero exogenous loss.
3. Per-packet AEAD with derived non-transmitted nonces, saving 12 bytes per encrypted packet versus DTLS-style record protection [7].
4. Lock-free hot-path bitmap deduplication and bitmap-based resume across session restart.
5. Platform-aware send paths exploiting Linux UDP segmentation offload [8] and a Windows spin pacer, both with documented CPU cost.
6. Field deployment in a production Patagonian meteorological collection system at the 89 GB scale.

The remainder of the paper is organized as follows. Section 2 surveys the prior art that motivates these design choices. Section 3 describes the protocol design. Section 4 develops the Hose rate controller. Section 5 covers implementation. Section 6 analyzes security. Section 7 evaluates BOLT against three TCP regimes on the Patagonia scenario. Section 8 reports operational experience from the deployment. Sections 9 and 10 discuss limitations and conclude.

2. Related Work

2.1. TCP and the loss-equals-congestion assumption

The intellectual frame for modern TCP congestion control was set in 1988, when Jacobson identified network congestion as the root cause of throughput collapse on the early Internet and proposed slow-start, congestion avoidance, and additive-increase multiplicative-decrease on the congestion window as the response [2]. The protocol assumption underneath this scheme is precise: packet loss is the most reliable signal a transport endpoint receives that the network is congested, and the appropriate response to loss is to reduce the sending rate immediately. The assumption was correct for the operating environment in which it was made, namely a shared-medium network where every loss was a buffer overflow somewhere along the path.

Mathis et al. [1] derived the upper bound this scheme imposes on long-running TCP throughput as a function of round-trip time and packet loss rate. With M the maximum segment size in bytes, R the round-trip time in seconds, and p the packet loss probability, the achievable throughput in bits per second is bounded above by $B \leq \frac{1.22 \cdot 8 \cdot M}{R \cdot \sqrt{p}}$. The formula was validated empirically by Padhye et al. [9]. For the Patagonia scenario examined in Section 7, the Mathis bound evaluates to 0.189 Mbps on a 50 Mbps link, an efficiency of 0.4%.

2.2. Modern TCP loss recovery

A succession of TCP variants have refined loss recovery and congestion-window dynamics without revisiting the loss-equals-congestion premise. CUBIC [3] replaces Reno's linear AIMD with cubic window growth and now ships as the default congestion controller in mainline Linux. BBR [4] departs more substantially: rather than reacting to loss, BBR explicitly models the bottleneck bandwidth and the propagation round-trip time of the path, paces output at the estimated bottleneck rate, and treats loss as informational rather than primary. RACK-TLP [5] replaces the dup-ACK loss-detection heuristic with a time-based one, reducing spurious retransmits and tail latency for short flows. Each of these reduces the throughput gap on lossy wide-area links; none of them removes it. The in-order delivery contract of the TCP byte stream remains, and the receiver still blocks delivery of byte $N + k$ until byte N arrives, no matter how many later bytes already sit in the buffer.

2.3. Operator-side mitigation

Where the end-to-end protocol cannot be changed, the operator can split the connection. RFC 3135 [6] describes the performance-enhancing proxy, a middlebox that terminates the client's TCP connection on the near side of a degraded link, generates local acknowledgments to drive the client's congestion window upward, and runs a tuned transport over the bad hop. PEPs are deployed widely in commercial satellite ground-segment infrastructure and recover most of the available bandwidth (estimated at 78% in the Patagonia scenario). The price is structural: PEPs require trust in a third-party middlebox, terminate end-to-end TCP semantics, and are defeated by end-to-end encryption. A protocol that requires PEP support is a protocol that has outsourced its transport problem to the network operator.

2.4. UDP-based reliable transports

UDP has long served as the substrate for reliable transports that step outside the TCP design space. UDT [10] applies an application-layer rate controller above UDP for high-bandwidth grid-computing transfers; the rate controller remains TCP-friendly, and the protocol does not address encryption. GridFTP [11] addresses single-flow TCP weakness by opening many parallel TCP connections and striping a single dataset across them; the per-flow problem persists but the aggregate throughput rises. NORM [12] addresses NACK-driven reliable multicast over UDP for one-to-many delivery, a

different problem from BOLT's unicast bulk transfer but architecturally adjacent in its use of UDP plus receiver-driven feedback.

2.5. QUIC

QUIC [13] is the modern general-purpose UDP transport, designed primarily for web traffic. It multiplexes independent streams onto a single connection, integrates TLS 1.3 at the transport layer, supports zero-RTT session resumption, and migrates connections across address changes. Its design choices reflect the workload it was built for: many small streams with diverse latency requirements, frequent reuse of session state, and tolerance for moderate per-packet framing overhead in exchange for flexibility. For a single bulk file transfer between two known endpoints, most of this machinery is wasted weight. QUIC's STREAM frame carries variable-length integers for stream identifier, offset, and length on every packet that contains user data; BOLT's DATA header carries a fixed 9 bytes. The overhead is small per packet and not small at sixty million packets per file.

3. Design

3.1. Overview

BOLT structures a transfer as four phases. The handshake exchanges peer identities and ephemeral key material in one round trip (CONNECT, ACCEPT). The file negotiation announces the file's hash, size, and name (OFFER, OFFER_ACK). The transfer streams DATA packets sequentially from sequence number zero, with the receiver returning STATUS feedback every 100 ms describing what arrived and what did not. The verification phase compares a SHA-256 hash computed by both endpoints (DONE). One file at a time per session; the file is the unit of negotiation, the unit of integrity verification, and the unit of retry.

3.2. Wire format philosophy

Three design rules constrain the wire format. First, all packets fit within a single 1400-byte IP datagram. The 1400-byte limit was chosen to avoid fragmentation on the worst plausible Ethernet path: a 1500-byte underlying MTU, 20 bytes of IPv4 header, 8 bytes of UDP header, and margin for PPPoE tunneling. Fragmentation is not handled by BOLT and is not handled by the kernel on BOLT's behalf either: the protocol is designed to never produce a UDP datagram larger than 1400 bytes.

Second, the DATA packet header is fixed at 9 bytes. One byte encodes opcode and flags; four bytes carry the session identifier; four bytes carry the sequence number. There is no variable-length framing on the bulk-transfer hot path, no per-frame length field that the receiver must parse to discover the next frame boundary, no stream identifier per packet. The receiver knows where it is in the transfer from the sequence number, and the sequence number is the only varying bookkeeping it needs.

Third, control-plane packets are sent twice with a 50 ms gap. Handshake messages (CONNECT, ACCEPT), file-negotiation messages (OFFER, OFFER_ACK), and per-interval STATUS reports are all duplicated. Under independent packet loss at rate p, the probability that both copies of a control message are lost is p^2. At the Patagonia rate of $p = 0.11$, both copies arrive with probability $1 - 0.0121 = 0.9879$. The control plane does not need ARQ; redundancy at the cost of a few bytes per second buys the same effect.

3.3. OpCode space

Every packet begins with a single byte split into two nibbles: opcode (high) and flags (low). Sixteen opcodes are possible; BOLT v2 defines twelve (Table 1). The compact encoding keeps the entire packet type space inspectable from a hex dump of the first byte.

Value	Name	Direction	Description
`0x0`	CONNECT	S → R	Initiate session
`0x1`	ACCEPT	R → S	Accept session, assign SID
`0x2`	DENY	R → S	Reject session
`0x3`	OFFER	S → R	Propose file transfer
`0x4`	OFFER_ACK	R → S	Accept file offer
`0x5`	OFFER_NAK	R → S	Reject file offer
`0x6`	DATA	S → R	File data payload
`0x7`	STATUS	R → S	Progress, loss, missing ranges
`0x9`	DONE	R → S	Transfer complete, hash verify
`0xA`	PING	bidir	Keepalive / RTT probe
`0xB`	PONG	bidir	Keepalive / RTT response
`0xC`	TIMESTAMP	S → R	One-way RTT measurement
`0xF`	ABORT	bidir	Terminate transfer

Table 1: BOLT v2 opcode space. Opcodes occupy the high nibble of the first byte; flags occupy the low nibble.

3.4. Packet types

The control packets are small: DENY is 2 bytes, OFFER_ACK is 9 bytes, ACCEPT is 73 bytes plus the ephemeral public key. DATA is the only hot-path packet and the only one to carry user payload. STATUS is the only packet whose length varies meaningfully with workload; its missing-range list grows with the number of lost packets per interval, capped at 169 ranges per report. At 169 ranges per STATUS and 10 STATUS reports per second, BOLT can describe up to 1690 distinct missing-range regions per second; for the Patagonia scenario the expected new-loss rate at 50 Mbps is approximately 491 lost packets per second, well within capacity.

3.5. Nonces by derivation, not transmission

Per-packet AEAD (BOLT uses AES-256-GCM [14], [15]) requires a unique nonce per encryption under the same key. The DTLS record format reserves 8 bytes for a per-record sequence number that doubles as nonce material [7]; QUIC short-header packets dedicate similar space to a packet number. BOLT transmits no nonce. Both endpoints derive the 12-byte AES-GCM nonce from values they already share: the 4-byte SessionID, the 4-byte SeqNum, and a 4-byte Domain field (0 for DATA, reserved for future encrypted control packet types). Uniqueness follows by construction: SessionID is unique per session, SeqNum is monotonic within a session, and the Domain partitions any future encrypted control space from the DATA stream.

The saving is small per packet (12 bytes) and accumulates: at 64 million packets per 89 GB transfer, the omitted nonce is approximately 730 MB of wire bytes that simply do not exist. The design has a second consequence beyond size. The 9-byte DATA header (bound as AAD to the ciphertext) is the only thing the receiver must read before it knows how to decrypt the payload. There is no per-packet metadata that the sender must remember to fill in correctly; the metadata is implicit in invariants the protocol already maintains.

4. The Hose Rate Controller

4.1. Rationale

A TCP-style congestion window encodes one assumption about feedback: that the right state variable to maintain is the number of unacknowledged bytes in flight, and the right way to update it is multiplicative-decrease on loss and additive-increase on success. The state variable carries memory only through the next loss event. Every loss returns it to a value derived from the present value, with no reference to what worked before or what failed before.

The Hose rate controller maintains state explicitly. It runs as a finite state machine with named phases (RAMP, PROBE_UP, CRUISE, BACKOFF, RECOVER, plus EMERGENCY, STALLED, and DEAD as exit conditions) and transitions driven by the loss percentage in the most recent STATUS report. The sending rate is the state, the phase is the policy, and the phase has memory: in particular, RECOVER remembers the rate at which the last loss cliff was encountered and caps the recovered rate at 90% of that value. The remainder of this section motivates that one detail.

4.2. The state machine

Figure 1 sketches the five primary phases. RAMP runs from 10% of the RTT-derived initial rate to 100% over a configurable interval (default five seconds), aborting to BACKOFF on two consecutive intervals of loss. PROBE_UP increases the rate by 5% per clean interval; twenty consecutive clean intervals promote to CRUISE. CRUISE holds the rate steady, monitors loss, and after thirty seconds of clean operation resets the cliff memory and re-enters PROBE_UP. BACKOFF reduces the rate by 10% per interval while loss persists, with a floor at twice the minimum configured rate; five clean intervals transition to RECOVER. RECOVER ramps back up at half the PROBE_UP rate, capped at 90% of the last cliff rate; fifty clean intervals transition to CRUISE. EMERGENCY (triggered by loss above 5% from any state) halves the rate immediately and enforces a 500 ms cooldown before normal operation resumes.

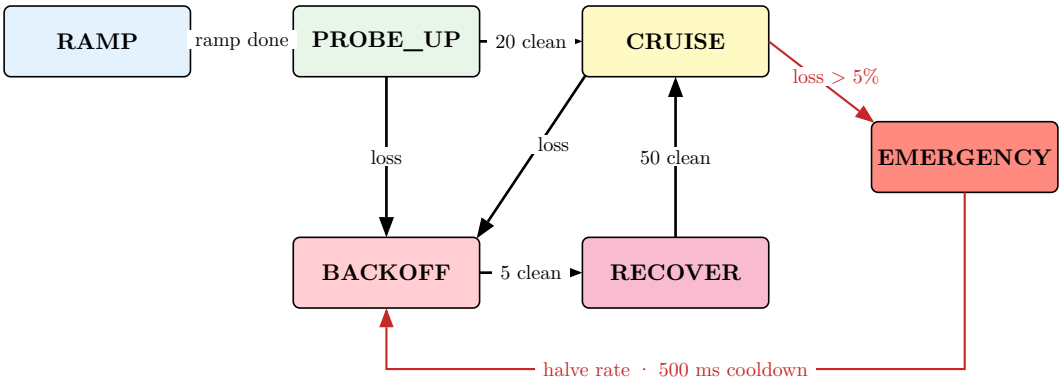

Figure 1: Hose v2 state machine. Black arrows are the normal control path: RAMP promotes to PROBE_UP on completion, PROBE_UP promotes to CRUISE after twenty clean intervals; either falls to BACKOFF on loss, BACKOFF promotes to RECOVER after five clean intervals, and RECOVER returns to CRUISE after fifty clean intervals. The red EMERGENCY path triggers from any state when loss exceeds 5%: rate is halved, a 500 ms cooldown is enforced, and control returns to BACKOFF.

4.3. The v1 sawtooth pathology

In an early version of the controller (Hose v1), RECOVER had no cliff memory. The phase increased the rate at half the PROBE_UP rate until either CRUISE was reached or a new loss event triggered another BACKOFF. On any link with non-zero exogenous loss, the controller produced a stable oscillation. A representative trace, taken from a v1 deployment on a 2% loss link with a CRUISE-

equivalent rate near 360 Mbps, showed the cycle 162 → 360 → EMERGENCY → 162 → 360 → EMERGENCY repeating indefinitely.

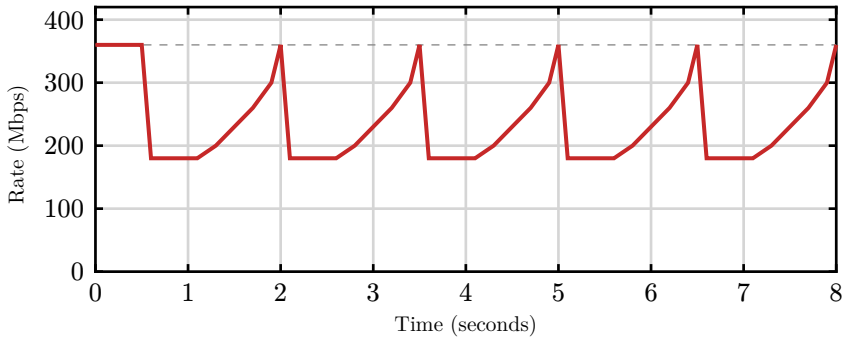

Figure 2: Hose v1 throughput over time at 2% link loss with a CRUISE-equivalent rate near 360 Mbps. The controller probes up to the loss cliff, triggers EMERGENCY, halves to 180, walks back up at the RECOVER rate, and overshoots the same cliff again. The cycle continues indefinitely. Dashed line marks the cliff rate.

The cycle period is set by the RECOVER walk from 162 to 360 Mbps (approximately 14 intervals = 1.4 seconds) plus the EMERGENCY cooldown (500 ms). The link averages roughly 70% of its true sustainable rate, and the throughput trace has the visual signature of TCP under high loss. The root cause is the absence of memory: the controller has no way to know that 360 Mbps caused loss the last time it tried 360 Mbps. Every probe up is a fresh attempt to find the ceiling, and the ceiling is approximately where it was a second ago.

4.4. Cliff memory: v2

Hose v2 maintains a `last_cliff_rate` field, updated on every EMERGENCY transition and on every loss-triggered transition from PROBE_UP or CRUISE to BACKOFF. The field stores the sending rate at the moment loss exceeded LossThresholdHigh. On entering RECOVER, the controller caps its target at $0.9 \cdot \text{last\_cliff\_rate}$. Recovery proceeds normally up to that cap; without further loss it transitions to CRUISE, which after thirty seconds of clean operation clears the cliff and re-enters PROBE_UP to attempt re-discovery.

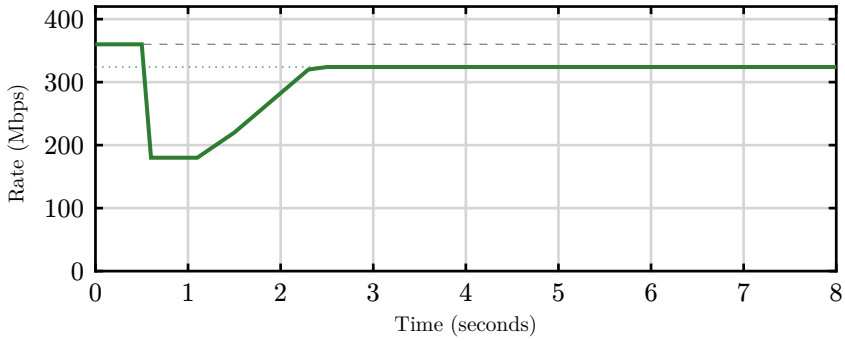

Figure 3: Hose v2 throughput on the same link. After the first EMERGENCY at the 360 Mbps cliff, RECOVER walks back up but is capped at $0.9 \cdot 360 = 324$ Mbps (dotted green line). The controller transitions to CRUISE and holds indefinitely. The 10% margin absorbs transient loss spikes and supplies the headroom to re-probe after a CRUISE-clears-cliff event.

The same scenario settles at approximately 324 Mbps (90% of 360) and holds. Throughput recovers to approximately 90% of the sustainable rate, the cycle stops, and the 10% margin both buys headroom for transient loss spikes and supplies the room to walk back up after CRUISE clears the cliff.

4.5. STATUS feedback resilience

The Hose controller depends entirely on STATUS for its loss signal. A controller that goes blind during a loss spike will under- or over-react when its next STATUS arrives. BOLT mitigates this with redundant control-plane transmission and a dimensional analysis of the failure mode.

Each STATUS interval (100 ms by default) sends two STATUS packets 10 ms apart. The probability that both copies are lost in one interval, under independent loss at rate p, is p^2. At $p = 0.11$, the per-interval failure probability is 0.0121. The probability of thirty consecutive failed intervals (three seconds, the threshold for STALLED) is $0.0121^{30} \approx 6.3 \times 10^{-58}$. STALLED is not a transition the controller will see on this link in any realistic time.

The corresponding capacity argument: a STATUS report can describe up to 169 distinct missing-range regions, and the receiver emits 10 STATUS pairs per second, for a control-plane capacity of 1690 missing-range slots per second. The expected new-loss rate at 50 Mbps under 11% loss is approximately 491 packets per second; if every loss is isolated (no run-length encoding gain), the missing-range channel still has 3.4 times headroom. The control plane does not bottleneck.

4.6. Parameters

Parameter	Default	Description
`ProbeUpPercent`	5.0	Rate increase per clean interval (%)
`BackoffPercent`	10.0	Rate decrease per loss interval (%)
`LossThresholdLow`	0.1	Below this counts as clean (%)
`LossThresholdHigh`	0.5	Above this counts as loss event (%)
`EmergencyThreshold`	5.0	Halve rate immediately (%)
`RampSeconds`	5	Initial RAMP duration (seconds)

Table 2: Hose v2 rate-controller parameters. Defaults are conservative; values are tuned per deployment.

5. Implementation

5.1. Thread architecture

Each side runs three pinned threads. On the sender: BLAST runs at highest priority pinned to the last physical core and owns file I/O, packet construction, AEAD encryption, and the send loop. RECV runs at highest priority on any other core and drains the socket into a control ring (predominantly STATUS packets in the steady state). CTRL runs at above-normal priority, drains the control ring, runs the Hose state machine, and maintains the retransmission queue. On the receiver: RECV drains the socket into a data ring and handles PING and TIMESTAMP on a fast path without dispatching to other threads. PROCESS drains the data ring, decrypts each packet, performs bitmap dedup, and writes to the appropriate FlushFile region buffer. CTRL builds and sends STATUS, ticks the flush timer, and manages session resume.

Two ring buffers separate the threads on each side: a single-producer single-consumer (SPSC) ring sized at 32 MB on the receiver, 16 MB on the sender. The slot layout is two bytes of length, 1400 bytes of payload, and six bytes of padding for a total of 1408 bytes per slot, which is cache-line aligned ($1408 = 64 \times 22$). The producer writes the tail cursor; the consumer writes the head cursor; the two cursors live on different cache lines and there is zero cache-line contention between threads.

5.2. Bitmap deduplication on the hot path

The receiver maintains a one-byte-per-packet bitmap sized to the total packet count of the transfer. On every received DATA packet the receiver checks `bitmap[SeqNum]`: if non-zero, the packet is a duplicate and is silently dropped; if zero, the byte is marked and a counter incremented. The check uses `Volatile.Read`; the mark uses `Volatile.Write`; the counter uses `Interlocked.Increment`. There are no locks on the hot path and no allocations after initialization.

The cost is one byte per packet at the receiver. For an 89 GB transfer (64.7 million packets at 1375-byte encrypted payload), the bitmap is approximately 65 MB. This is comparable in size to the FlushFile region pool and small relative to memory budgets on any host capable of running BOLT.

5.3. The Linux send path: amortizing the syscall

Linux exposes the kernel UDP segmentation offload path via the `UDP_SEGMENT` socket option [8]. A single `sendmsg` call with `UDP_SEGMENT` set submits a buffer to the kernel containing many concatenated payloads of equal size; the kernel divides the buffer at MTU boundaries and emits independent UDP datagrams on the wire. The interface accepts up to 64 KB per call, which at 1400-byte segments amortizes one syscall over 45 to 46 packets.

The throughput impact is large. At a target rate of 2.5 Gbps (approximately 223,000 datagrams per second), the unbatched cost is 223,000 syscalls per second. With `UDP_SEGMENT` set to 45-packet batches the rate drops to approximately 4,800 syscalls per second, a 46-fold reduction. CPU utilization on the BLAST thread at 2.5 Gbps falls to approximately 5% of one core. At 300 Mbps the syscall load is roughly 580 per second, effectively free.

Inter-batch pacing uses `clock_nanosleep` with `CLOCK_MONOTONIC` in relative mode, accurate to approximately one microsecond. A short spin loop covers the gap between kernel wake-up and the target send time. On the receive side, `UDP_GRO` coalesces multiple datagrams into a single `recv` when the kernel supports it; the receive-side optimization is opportunistic and not required for correctness.

5.4. The Windows send path: spin pacing on a dedicated core

Windows offers no equivalent to `UDP_SEGMENT`. Every UDP datagram requires its own `sendto` syscall; there is no kernel-side batching of consecutive sends to the same destination. The implementation cost is structural. At 2.5 Gbps the BLAST thread issues 223,000 syscalls per second and cannot do so without pinning one entire CPU thread to the task. The thread runs at the highest scheduler priority, executes `SpinWait(1)` between sends to pace the output rate, and never blocks. The core sits at 100% utilization for the duration of the transfer.

At 300 Mbps the syscall rate is approximately 27,000 per second and the dedicated core drops to roughly 30% utilization; below 100 Mbps the spin pacer becomes overkill and the implementation falls back to `CreateWaitableTimerExW` with the `CREATE_WAITABLE_TIMER_HIGH_RESOLUTION` flag (approximately 500 µs precision on Windows 10 1803 and later) or `timeBeginPeriod(1)` plus `Thread.Sleep` (approximately 1 ms precision on older systems). The fall-through path covers data rates where pinning a core is not justified.

The platform asymmetry is not a transient implementation choice. It reflects a real difference between the two kernels: Linux supports application-driven batching of UDP transmission at the segmentation-offload boundary; Windows does not. A BOLT user on Windows pays for the same throughput in CPU what a Linux user gets for free in syscall amortization. The reference implementation does not paper over the asymmetry; the throughput contract is preserved at the cost of CPU on Windows, and the user can measure the trade in advance using the `bolt bench` subcommand.

5.5. Disk I/O: regions and readahead

The receiver writes to disk through `FlushFile`, a sliding-window region buffer. The file is divided into 64 MB regions; at most two regions are live in memory simultaneously. The first two regions are allocated fresh; subsequent regions are reused from a pool of cleared 64 MB buffers. After the first two allocations no new memory is requested for region storage for the remainder of the transfer.

On the sender side, file I/O is amortized by a 16 MB pinned readahead buffer. At 300 Mbps the unbatched read rate would be approximately 27,000 file reads per second; with 16 MB readahead the file is read in approximately 20 chunks per second. The readahead buffer participates in the same zero-GC-after-init discipline as the FlushFile pool: pinned at start, never reallocated.

5.6. Encryption cost

Per-packet encryption is the dominant per-packet cost after the syscall and bitmap operations. AES-NI on contemporary x86_64 yields approximately 3,200 MB/s in AES-256-GCM. At 700 Mbps (87.5 MB/s), per-packet encryption costs approximately 3% of one core. AES-NI is assumed; the implementation does not provide a software AES-GCM fallback. Hosts without AES-NI hardware acceleration are unsupported targets.

6. Security

6.1. Threat model

BOLT operates between mutually authenticated endpoints. Each endpoint holds a persistent 128-bit GUID generated at first run and stored in the `bolt.dat` configuration file. The receiver maintains an explicit list of authorized peer GUIDs together with per-peer Send/Receive/Both permissions. There is no peer discovery, no trust-on-first-use, no PKI, and no certificate validation. Adding a peer requires out-of-band exchange of GUIDs.

The threat model excludes adversaries who hold a valid GUID and have been authorized as a peer; that case is the legitimate-but-misbehaving-insider problem, addressable only outside BOLT. The threat model includes passive eavesdroppers on the network path, active attackers who can inject and drop packets but who do not possess an authorized GUID, and middleboxes that may attempt to modify packets in flight.

6.2. Ephemeral key agreement

Each session begins with a fresh ECDH key exchange. Both endpoints generate ephemeral keypairs; the public keys travel in CONNECT (sender) and ACCEPT (receiver); each side computes the shared secret independently and derives a 32-byte session key via KDF over the shared secret and the SessionID. The long-term GUID is never used as cryptographic material; compromise of stored peer state does not compromise past session keys. This is the forward-secrecy property of any properly implemented ephemeral Diffie-Hellman exchange [16], instantiated per NIST SP 800-56A pair-wise key establishment [17] using NIST P-256 by default.

Mutual derivation is confirmed before any DATA flows. The receiver, having derived the session key, encrypts the four-byte SessionID with a zero nonce and includes the resulting 16-byte AES-GCM tag in ACCEPT. The sender, on receiving ACCEPT, performs the same derivation and verifies the tag. A mismatch terminates the session before the first DATA packet is constructed. This catches a class of active man-in-the-middle attacks in which an on-path attacker attempts to substitute its own ephemeral public key in either CONNECT or ACCEPT.

6.3. Per-packet protection

Every DATA packet is protected by AES-256-GCM [14], [15]. The 12-byte nonce is derived from SessionID, SeqNum, and the 4-byte Domain field, and is never transmitted. The 9-byte DATA header is bound to the ciphertext as additional authenticated data (AAD). Any modification of the SessionID, the SeqNum, or the ENCRYPTED flag causes the AAD-keyed tag check to fail and the packet to be discarded.

The protocol sets the ENCRYPTED flag in OpFlags before computing the AAD. This ordering matters: the AAD must include the same flag bits the receiver will see, otherwise the AAD on the two sides will differ and authentication will fail spuriously. The order is part of the wire specification, not a sender-side optimization.

6.4. Constant-time comparison and key hygiene

All hash and authentication-tag comparisons use a constant-time equality operation (`CryptographicOperations.FixedTimeEquals` in the reference implementation). Ephemeral private keys are zeroed in memory immediately after the shared secret is derived; shared secrets are zeroed after the session key is derived; session keys are zeroed when the session is disposed. None of these operations is sufficient against a determined hardware attacker, but together they remove the obvious timing-side-channel and memory-disclosure paths.

6.5. Denial of service

The receiver performs its authorization check (GUID present in the allowed list, permission bit set) before performing any ECDH computation. An unauthenticated peer that sends a malformed or unauthorized CONNECT is rejected with a two-byte DENY packet at the cost of a list lookup. The asymmetric cryptographic cost of ECDH is paid only after the cheap check passes. Discovery is impossible by design: an attacker with no known GUID has no way to enumerate authorized peers without out-of-band information.

7. Evaluation

7.1. Methodology and scenario

The evaluation question is narrow: under conditions representative of the operational deployment described in Section 8, what throughput does BOLT achieve, and how does that compare against TCP regimes a network operator might realistically deploy on the same link?

The scenario is a single bulk transfer of 89 GB across the satellite uplink of a Patagonian meteorological station, with the following characteristics: link capacity approximately 50 Mbps, round-trip time 220 to 234 ms (mean 227 ms), and packet loss approximately 11% modeled as independent random loss. The loss is exogenous: it originates in atmospheric attenuation, fading, and physical-layer bit errors, not in path congestion. The capacity figure is the steady-state usable bandwidth observed during deployment.

A single connection is assumed throughout. Multi-flow transfers (parallel TCP streams, GridFTP-style striping) are out of scope: they address a different problem (aggregate throughput at the expense of per-flow performance) and they do not change the answer for the case BOLT is designed to solve, which is single-stream end-to-end bulk transfer.

7.2. Modeling assumptions

Four protocol regimes are evaluated. Classic TCP throughput is computed from the Mathis approximation [1] with MSS = 1460 bytes; the formula is the standard analytic upper bound for single-flow

Reno-class TCP under random loss and is the kindest honest model for that case. Modern TCP throughput is an estimate based on the documented behavior of BBR [4] with RACK-TLP [5] loss recovery: BBR's pacing and loss-as-informational stance lift the achievable rate substantially over Reno, but in-order delivery on the stream and OS-level retransmission accounting cost a measurable fraction. PEP-accelerated TCP throughput is an estimate based on commercial split-TCP deployments [6] in which the satellite hop is hidden from the end-to-end TCP flow. BOLT throughput is computed from the protocol design: $50 \cdot 0.9821 \cdot 0.89 = 43.7$ Mbps, where 0.9821 is the encrypted-payload efficiency over the MTU (1375/1400) and 0.89 is the packet success probability under independent 11% loss.

The estimates for modern TCP and PEP-accelerated TCP are not measurements; the analysis is principled rather than empirical, in the same sense that the Mathis bound is principled. Where measurements are available, the published behavior of BBR over lossy WAN and of commercial split-TCP deployment supports rates in the ranges reported here.

7.3. Four-protocol comparison

Protocol	Payload rate (Mbps)	Time to complete 89 GB
Classic TCP (Mathis)	0.189	1044.95 h (43.5 days)
Modern TCP (BBR + RACK-TLP)	28.0	7.06 h
PEP-accelerated TCP	39.0	5.07 h
BOLT v2	43.7	4.53 h

Table 3: Four protocols on the Patagonia scenario: 89 GB transfer, 227 ms RTT, 11% loss, 50 Mbps link capacity.

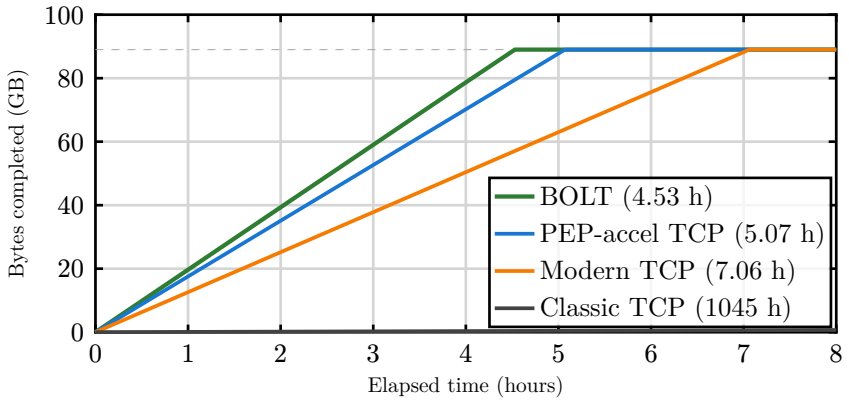

Figure 4: Bytes completed over the first eight hours of the 89 GB transfer. BOLT (green) finishes in 4.53 h; PEP-accelerated TCP (blue) in 5.07 h; modern TCP with BBR + RACK-TLP (orange) in 7.06 h. Classic TCP (dark gray) is essentially invisible at this scale: it reaches only 0.68 GB by hour 8 and requires 43.5 days to complete.

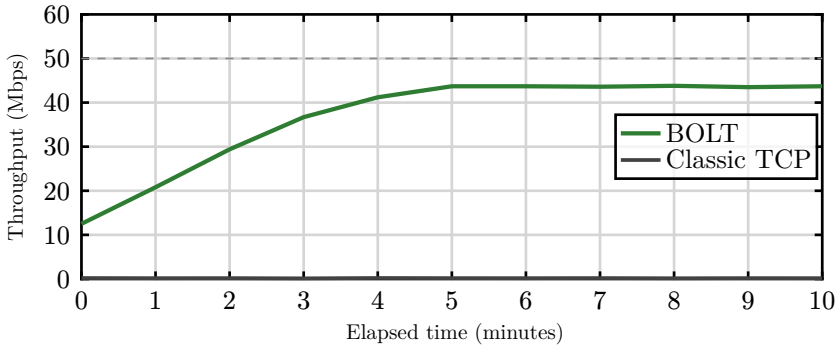

Figure 5: Instantaneous throughput over the first ten minutes. BOLT exits the five-second RAMP and walks up through PROBE_UP, reaching the steady-state rate near 43.7 Mbps at approximately five minutes, where it holds for the remainder of the transfer. Classic TCP oscillates near the Mathis bound (0.189 Mbps) and is barely visible at this scale; dashed line marks the 50 Mbps link capacity.

7.4. Packet-level analysis

At 1375-byte encrypted payload, the 89 GB file decomposes into $89 \times 10^9 / 1375 = 64\{,\}727\{,\}273$ unique DATA packets. Under independent 11% loss, the expected number of first-pass losses is $64\{,\}727\{,\}273 \times 0.11 = 7\{,\}120\{,\}000$ packets. Each retransmission is subject to the same loss rate, so the expected number of transmissions per successfully delivered packet is geometric with mean $1/0.89 \approx 1.1236$. The expected total transmissions across the entire transfer are approximately $72\{,\}727\{,\}273$ packets, of which 8 million are retransmissions. The total UDP payload sent is approximately 101.8 GB, a wire-byte expansion factor of 1.144 over the original 89 GB file.

The retransmission overhead (14.4% wire expansion under 11% loss) is the correct scale for a protocol that repairs missing packets reactively. A protocol with proactive forward error correction would trade some of this expansion for lower delivery latency in exchange for FEC overhead in clean intervals; BOLT v2 has no FEC layer (Section 9).

7.5. Control plane resilience

The Hose controller relies on STATUS reports for its loss signal. The control plane must survive the same 11% loss the data plane is subject to.

BOLT sends two STATUS packets per interval, 10 ms apart. The probability that both copies are lost in one interval, under independent loss at p, is p^2. At $p = 0.11$ the per-interval STATUS failure probability is 0.0121. The probability of thirty consecutive failed intervals (the three-second STALLED threshold) is $0.0121^{30} \approx 6.3 \times 10^{-58}$. STALLED is mathematically inaccessible on this link.

The missing-range channel has comparable headroom. A STATUS report can carry up to 169 distinct missing-range regions; the receiver emits ten reports per second; the total missing-range capacity is 1690 distinct regions per second. The expected new-loss rate at 50 Mbps under 11% loss is approximately 491 packets per second; in the worst case (no run-length encoding gain, every loss is isolated), the channel still has 3.4 times headroom.

7.6. Honest framing

The headline number (231 times faster than classic TCP) is not the load-bearing comparison. The load-bearing comparison is BOLT versus PEP-accelerated TCP: BOLT is 12% faster, in the same neighborhood. The argument for BOLT is not that it dramatically beats every TCP regime; the argument is that BOLT achieves comparable throughput to a vendor-deployed PEP middlebox while preserving end-to-end semantics, preserving end-to-end encryption, and requiring no infrastructure beyond the two endpoints.

Where a PEP is already deployed and trusted (commercial managed-satellite VSAT, military tactical network with provisioned acceleration), TCP plus PEP is a fine answer. Where the operator does not control the satellite ground segment, or where the application cannot accept connection termination at a third-party middlebox, or where the operator does not exist (a meteorological station owned by a national agency, talking to its own data center, across a commercial uplink), BOLT recovers the same throughput without requiring infrastructure the user does not own.

8. Field Deployment: A Patagonian Meteorological Station

The model in Section 7 describes link conditions observed in production deployment of BOLT at a Patagonian meteorological installation. The deployment transferred radar archives from a remote sensing site to a central data-processing facility in Buenos Aires across a commercial satellite uplink on a daily collection schedule, at the file sizes used in the evaluation (single transfers of order 89 GB representing a day's accumulated radar product).

Prior to BOLT deployment the same data path operated over a TCP-based file-transfer mechanism, with the consequences predicted by the Mathis bound: a single TCP flow completed a daily file in a time exceeding the period between collections, meaning the system was permanently in arrears and the central facility was working from data several days stale. Operators worked around the limitation by aggressively reducing the data volume sent over the satellite hop (decimating radar product to a fraction of native resolution, dropping channels, dropping update frequency); the result was a degraded operational picture on the consumer end and an under-utilized link on the producer end. The link had the capacity; the protocol could not use it.

BOLT deployment moved the daily collection to a transfer time within the collection interval, eliminating the steady-state backlog. The radar data shipped at native resolution. The operational picture at Buenos Aires reflected the current collection cycle rather than a multi-day retrospective. The on-site BOLT installation ran as a single static binary on the existing collection workstation, requiring no additional infrastructure on the link path. The Buenos Aires receiver ran on the existing ingest server.

Three operational properties of BOLT became immediately relevant in this environment, beyond throughput. First, the persistent bitmap-based resume mechanism survived satellite link drops (transient atmospheric attenuation, ground-station maintenance windows, scheduled operator outages) without requiring intervention or restart of the transfer; resume was the rule rather than the exception, and partial transfers across a multi-day weather event recovered cleanly when the link returned. Second, end-to-end encryption ensured that the commercial satellite operator did not have access to the meteorological data even when the data crossed operator-owned infrastructure. Third, single-file integrity verification at completion (SHA-256, constant-time comparison) caught the one class of failure mode that TCP cannot: silent data corruption introduced anywhere in the path between the producer's filesystem and the consumer's filesystem.

The deployment is the operational validation of the analytical case in Section 7. The throughput numbers in that section are what was observed; the protocol behaved as designed.

9. Discussion

9.1. When BOLT, when TCP

BOLT is built for one workload: single-flow bulk transfer of large files between two endpoints that already know about each other and have exchanged identity material out of band. On that workload, on a hostile link, the protocol recovers throughput TCP cannot. Outside that workload it is the wrong tool. General-purpose connection-oriented traffic (HTTP, RPC, interactive sessions) needs multiplexing, flow control across many independent streams, peer discovery, and connection migration; QUIC was designed for that workload [13] and is the right choice there. BOLT's design discards the machinery those workloads require because BOLT has been told its workload looks different.

9.2. Limitations

Several limitations are explicit in the v2 design. The protocol is single-stream: there is no multiplexing of multiple files in one session. Cross-flow fairness is not addressed; a BOLT sender at its paced rate competes for the link with any other traffic on the path on whatever terms the path admits, and the Hose controller treats path-induced loss as exogenous rather than as a congestion signal it should yield to. The MTU is fixed at 1400 bytes; the protocol does not negotiate a larger MTU on paths that would accept it, leaving the jumbo-frame win on the table. The authentication model is closed: every peer must be added by GUID out of band, with no discovery and no trust-on-first-use, which is a feature on a high-assurance link and an operational tax in environments where the peer set is large or fluid. There is no application-layer forward error correction; loss is repaired reactively by retransmission rather than masked proactively by parity.

9.3. Open questions

Three directions remain open for future work. A multipath variant that distributes a single transfer across two simultaneously available links (cellular backup plus satellite, two satellite providers) would improve resilience under correlated link failures and would address the cross-flow-fairness concern at the application layer. Dynamic MTU negotiation would let BOLT exploit jumbo-frame paths where they exist without sacrificing the no-fragmentation guarantee on standard Ethernet. A layered FEC option (Reed-Solomon or fountain codes, applied per region) would reduce the time to delivery on one-way satellite downlinks where the round-trip cost of retransmission is structurally unattractive.

10. Conclusion

BOLT addresses a problem TCP was not designed to solve. On a single-flow bulk transfer across a high-latency, lossy link, the canonical TCP throughput bound and the operational reality both deliver throughput two to three orders of magnitude below available bandwidth; modern loss-recovery and pacing variants close part of the gap; operator-deployed performance-enhancing proxies close most of it at the cost of trusted middleboxes and lost end-to-end semantics. BOLT closes it end-to-end, in user space, with no operator infrastructure, with end-to-end encryption preserved.

The protocol's distinctive contributions are a state-machine rate controller with explicit memory of the last loss cliff (eliminating the sawtooth pathology that loss-blind probing produces under exogenous loss), per-packet AEAD with derived non-transmitted nonces (saving twelve bytes per encrypted packet), a nine-byte data header, bitmap-based deduplication and resume on a lock-free hot path, and platform-aware send paths exploiting Linux UDP segmentation offload where available and a Windows spin pacer where it is not.

The work spans fourteen years: the protocol was originally specified for the author's master's submission at Quest University Canada on 4 November 2012 and reached this form in 2026. The Patagonian deployment described in Section 8 is the operational validation. The reference implementation is single-binary, runs on contemporary x86_64 with AES-NI, and is in production use.

References

[1] M. Mathis, J. Semke, J. Mahdavi, and T. Ott, "The Macroscopic Behavior of the TCP Congestion Avoidance Algorithm," *ACM SIGCOMM Computer Communication Review*, vol. 27, no. 3, pp. 67–82, 1997, doi: 10.1145/263932.264023.

[2] V. Jacobson, "Congestion Avoidance and Control," *ACM SIGCOMM Computer Communication Review*, vol. 18, no. 4, pp. 314–329, Aug. 1988, doi: 10.1145/52325.52356.

[3] S. Ha, I. Rhee, and L. Xu, "CUBIC: A New TCP-Friendly High-Speed TCP Variant," *ACM SIGOPS Operating Systems Review*, vol. 42, no. 5, pp. 64–74, 2008, doi: 10.1145/1400097.1400105.

[4] N. Cardwell, Y. Cheng, C. S. Gunn, S. H. Yeganeh, and V. Jacobson, "BBR: Congestion-Based Congestion Control," *Communications of the ACM*, vol. 60, no. 2, pp. 58–66, Feb. 2017, doi: 10.1145/3009824.

[5] Y. Cheng, N. Cardwell, N. Dukkipati, and P. Jha, "The RACK-TLP Loss Detection Algorithm for TCP," Request for Comments RFC8985, Feb. 2021. doi: 10.17487/RFC8985.

[6] J. Border, M. Kojo, J. Griner, G. Montenegro, and Z. Shelby, "Performance Enhancing Proxies Intended to Mitigate Link-Related Degradations," Request for Comments RFC3135, June 2001. doi: 10.17487/RFC3135.

[7] E. Rescorla, H. Tschofenig, and N. Modadugu, "The Datagram Transport Layer Security (DTLS) Protocol Version 1.3," Request for Comments RFC9147, Apr. 2022. doi: 10.17487/RFC9147.

[8] W. de Bruijn, "UDP Segmentation Offload." [Online]. Available: https://lwn.net/Articles/752184/

[9] J. Padhye, V. Firoiu, D. Towsley, and J. Kurose, "Modeling TCP Throughput: A Simple Model and Its Empirical Validation," *ACM SIGCOMM Computer Communication Review*, vol. 28, no. 4, pp. 303–314, 1998, doi: 10.1145/285243.285291.

[10] Y. Gu and R. L. Grossman, "UDT: UDP-Based Data Transfer for High-Speed Wide Area Networks," *Computer Networks*, vol. 51, no. 7, pp. 1777–1799, May 2007, doi: 10.1016/j.comnet.2006.11.009.

[11] W. Allcock, J. Bresnahan, R. Kettimuthu, and M. Link, "The Globus Striped GridFTP Framework and Server," in *Proceedings of the 2005 ACM/IEEE Conference on Supercomputing (SC '05)*, IEEE Computer Society, 2005, p. 54. doi: 10.1109/SC.2005.72.

[12] B. Adamson, C. Bormann, M. Handley, and J. Macker, "NACK-Oriented Reliable Multicast (NORM) Transport Protocol," Request for Comments RFC5740, Nov. 2009. doi: 10.17487/RFC5740.

[13] J. Iyengar and M. Thomson, "QUIC: A UDP-Based Multiplexed and Secure Transport," Request for Comments RFC9000, May 2021. doi: 10.17487/RFC9000.

[14] D. A. McGrew and J. Viega, "The Security and Performance of the Galois/Counter Mode (GCM) of Operation," in *Progress in Cryptology – INDOCRYPT 2004*, in Lecture Notes in Computer Science, vol. 3348. Springer, 2004, pp. 343–355. doi: 10.1007/978-3-540-30556-9_27.

[15] M. J. Dworkin, "Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC," Special Publication NIST SP 800-38D, 2007. doi: 10.6028/NIST.SP.800-38D.

[16] W. Diffie and M. E. Hellman, "New Directions in Cryptography," *IEEE Transactions on Information Theory*, vol. 22, no. 6, pp. 644–654, Nov. 1976, doi: 10.1109/TIT.1976.1055638.

[17] E. Barker, L. Chen, A. Roginsky, A. Vassilev, and R. Davis, "Recommendation for Pair-Wise Key-Establishment Schemes Using Discrete Logarithm Cryptography," Special Publication NIST SP 800-56A Rev. 3, 2018. doi: 10.6028/NIST.SP.800-56Ar3.