Relay — Secure Networked Messaging

Technical Design, Security, and Reliability Report

August 2026 · github.com/chasssis/Relay

Charles Stevens — author & developer

1 Project Overview

Relay is a secure messaging system that provides authenticated, confidential, directed messaging with verifiable delivery for a small, controlled network such as a lab group or a team on one LAN. The problem it solves is concrete: participants on different hosts need to exchange private, directed messages and *know what happened to each one* — accepted, delivered, read, or failed — without trusting the network path between them.

The system is one server and two interchangeable clients: a terminal client speaking length-framed JSON over TLS, and a browser client speaking the same JSON protocol over secure WebSockets (WSS). Both transports feed a single routing core, so every participant sees the same identities, roster, directed messages, replies, and delivery/read acknowledgements regardless of client type. Its intended users are small trusted groups — the reference demonstration uses three simultaneous participants across two hosts — rather than the open internet; the server host is trusted, and the threat model says so explicitly.

Major functionality: password authentication against salted `scrypt` records with temporary lockout; a live roster; directed messages with replies; a per-message outcome ledger (accepted → delivered → read, or failed with a stable reason code); heartbeats and bounded reconnection; a structured security audit log; and an optional read-only, loopback-only observability dashboard. The project began as an extension of a small open-source socket-chat tutorial and was substantially redesigned and expanded — protocol, authentication, security controls, routing, WebSocket transport, browser client, tooling, and test suite — into the v3 system described here.

2 Final System Architecture

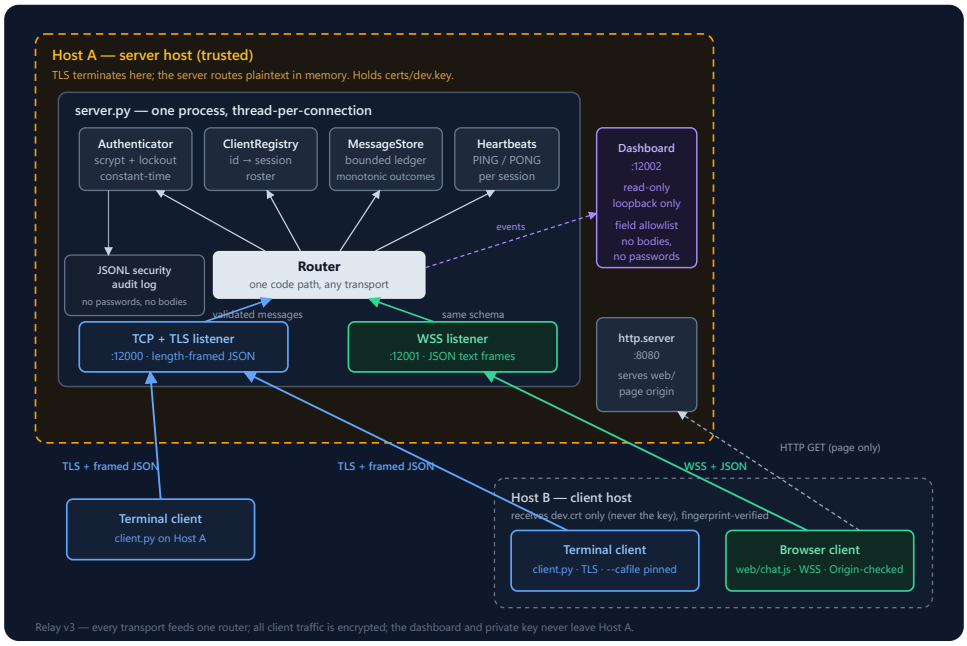

Figure 1: System architecture. Two transport listeners decode their framing and hand identical validated messages to one router; the amber dashed border is the trust boundary.

Figure 1 shows the final architecture. The server is a single Python process with two listeners: TCP+TLS on port 12000 for terminal clients and WSS on port 12001 for browsers. Each listener does nothing but transport work —

framing, TLS, WebSocket negotiation — and hands schema-validated messages to a shared **router**, which owns all application logic and consults four components: the **authenticator** (scrypt verification, lockout), the **client registry** (connection-scoped identities and the roster), the **message store** (a bounded ledger of message outcomes), and per-session **heartbeats**. A JSONL audit log records authentication events; an optional dashboard binds loopback only. Message state is in-memory and bounded by design — a live messaging system, not a store-and-forward mailbox — and credentials rest in a file of salted scrypt records managed by a CLI tool.

Why client/server. Directed messaging with authoritative delivery outcomes needs a single point that can verify identity once and stamp it on every routed message, order roster changes, and adjudicate each message's outcome. Peer-to-peer was considered and rejected: every pair of peers would need its own authenticated encrypted channel plus NAT traversal, and the properties Relay treats as central — one authoritative roster, one outcome ledger, replay rejection — have no natural home without an authority. A star topology gives one TLS identity to verify, one place to enforce protocol strictness, and one process to observe during failure testing.

Alternatives considered. An `asyncio` event loop was weighed against thread-per-connection; threads won because blocking I/O composes more simply with Python's `ssl` module and is easier to reason about under fault injection. The classic weakness — one stalled peer blocking others — is addressed head-on: a kernel-level `SO_SNDTIMEO` write deadline (10 s) on every socket, a global state lock *never held during I/O*, and per-session write locks. A browser-only system (one WSS transport) would have been simpler, but keeping raw TCP demonstrates explicit length-prefix framing, and forcing both transports through one router proved the protocol transport-independent. TCP was chosen over UDP because every message — credentials, messages, acknowledgements — needs reliable ordered delivery, and TLS, central to the security design, runs over a stream.

3 Protocol Design

Relay v3 is a custom application-layer protocol: UTF-8 JSON objects, framed on TCP by a four-byte big-endian length prefix and carried natively in WSS text frames. Validation is strict and directional — separate client and server schemas, unknown fields rejected, exact types enforced — with hard limits: 65,536-byte frames, 4,096-character bodies, 32-character usernames, 128-character correlation IDs, 1,000 roster entries. Eleven message types are documented one-by-one (purpose, required fields, responses, error conditions) in the protocol specification (`docs/protocol-spec.md`); a parity test keeps that catalog equal to the code's schema tables in both directions.

Message	Direction	Purpose
`AUTH`	client ⟶ server	Authenticate with username and password (one attempt, 10 s deadline)
`AUTH_SUCCESS`	server ⟶ client	Confirm identity; assigns connection-scoped ID (`alice#1024`)
`CLIENT_LIST`	server ⟶ client	Versioned roster snapshot on every membership change
`DM`	both	Directed message (client-sent), relayed with server-stamped sender
`DM_OUTCOME`	server ⟶ client	Monotonic outcome for a sent DM: `ACCEPTED` / `DELIVERED` / `READ` / `FAILED` +code
`ACK_READ`	client ⟶ server	Recipient marks a received DM read (authorized per recipient)
`PING` / `PONG`	server ⇄ client	Liveness probe: 15 s interval, 10 s answer deadline
`EXIT`	client ⟶ server	Clean disconnect
`ERROR`	server ⟶ client	Structured failure from an 11-code catalog, with correlation ID
`AUTH_FAILED`	server ⟶ client	Legacy shape, validated but never emitted by the v3 server

Table 1: Message catalog. Failures use stable codes (e.g. `DUPLICATE_MESSAGE_ID` , `UNKNOWN_RECIPIENT` , `HEARTBEAT_TIMEOUT` , `DELIVERY_FAILED`), parity-tested against the code.

Example exchange — the wire format is exactly this JSON:

```
C→S  {"type": "AUTH", "protocol_version": 3, "request_id": "a1", "username": "alice",
"password": "..."}
S→C  {"type": "AUTH_SUCCESS", "id": "alice#1024", "server_time":
"2026-08-01T18:02:11.482100Z", "server_seq": 1}
C→S  {"type": "DM", "id": "m1", "to": "bob#1025", "body": "hello", "reply_to": null}
```

```
S→C  {"type": "DM_OUTCOME", "msg_id": "m1", "outcome": "DELIVERED", "code": null, "detail":
null, "retryable": false, "server_time": "...", "server_seq": 4}
```

Protocol flow. Figure 2 traces one full session. Establishment is transport-level — TCP connect plus TLS handshake, or the WSS upgrade — and buys nothing by itself: the server allows exactly one `AUTH` within 10 seconds, then either assigns a connection-scoped identity or closes. Data exchange is directed `DM`s with the outcome ledger reporting progress back to the sender; termination is a clean `EXIT` or any fatal condition, both converging on one disconnect path that evicts the session and broadcasts a new roster.

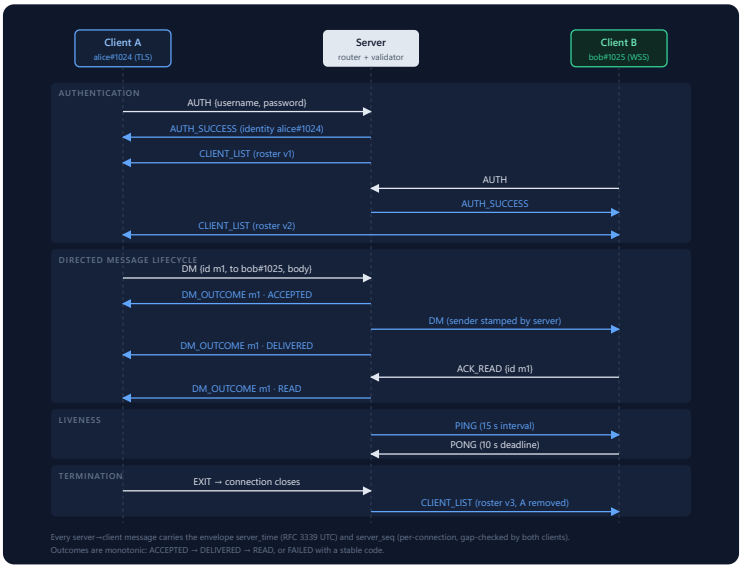

Figure 2: Protocol sequence: authentication, the directed-message outcome lifecycle, heartbeats, and termination.

Changes since v1. The v1 system (July 2026) already carried TLS/WSS, scrypt authentication, and delivery/read acknowledgements; its stated known limit was that silent half-open connections could linger. The protocol has since been extended four ways, each closing a named gap:

- **Heartbeats** (`PING` / `PONG`) — new message pair; a missed 10-second deadline produces a correlated fatal `HEARTBEAT_TIMEOUT` and eviction, directly closing v1's half-open-connection limitation.
- **Server envelope** — every server message now carries `server_time` (RFC 3339 UTC) and `server_seq`, a per-connection counter both clients gap-check, so a dropped or reordered server write is a detected protocol failure instead of silent divergence.
- **Unified outcome ledger** — delivery and read receipts became one monotonic `DM_OUTCOME` message with a retryable/fatal `FAILED` arm and stable reason codes — what makes honest failure reporting (§5) expressible on the wire.
- **Replay rejection** — duplicate `message_id`s are refused while retained in the bounded history, and the error catalog was pinned to eleven codes with a parity test so documentation and code cannot drift.

The test suite grew from 101 to 425 tests across the same period.

4 Security Design

Authentication. Username/password over the encrypted channel only. Credentials are stored as salted scrypt records ($N=2^{14}$, r=8, p=1, 16-byte random salt) and verified with a constant-time comparison. Scrypt was selected over fast hashes because it is memory-hard (SHA-256 alone is brute-forceable at GPU speed), and over anything custom because it ships vetted in Python's standard library. Three choices close the classic side channels: unknown usernames pay the same dummy-hash cost as real ones (timing), every failure returns the same generic fatal `ERROR` with code `AUTH_FAILED` (response text), and each connection gets exactly one attempt within a 10-second deadline (throttling). The router stamps the authenticated identity onto every routed message and ignores any client-supplied sender field,

so a client cannot claim to be someone else; identities are connection-scoped (`name#1024`), so a public ID cannot be replayed into an old conversation after a disconnect.

Confidentiality. TLS for terminal clients and WSS for browsers — chosen over message-level encryption because transport security protects *everything* (credentials, bodies, acknowledgements, the roster) with one well-understood mechanism. Clients pin the development certificate (`--cafile`) and verify its subject alternative names; the browser's page origin is checked against an explicit allowlist, defeating cross-site WebSocket hijacking. The development plaintext mode fails closed: `--insecure` is refused unless every address the bind host resolves to is loopback, and the browser client has no plaintext override at all for non-loopback hosts — a page can be opened by anyone, so a control that disables encryption is not offered.

Integrity. In transit, TLS/WSS record-layer authentication rejects tampered traffic on every supported path. On top of that, strict directional schemas discard malformed application data, the `server_seq` gap check detects dropped or reordered server writes, and duplicate-ID rejection prevents accepted messages from being silently re-applied.

Asset	Principal threats	Mitigations
Credentials	Password theft, guessing, timing probes	scrypt + salt, constant-time compare, dummy hashing, generic failures, 30 s lockout after 5 failures, audit log
Messages & receipts	Eavesdropping, tampering, replay	TLS/WSS everywhere, pinned certificate, strict schemas, duplicate-ID rejection, server-stamped sender
Sessions & roster	Impersonation, hijacked browser sockets	Connection-scoped identities, per-recipient `ACK_READ` authorization, WebSocket Origin allowlist
Server availability	Malformed input, resource exhaustion, stalled peers	Hard size limits, bounded stores, auth deadline, heartbeat eviction, 10 s write deadline

Table 2: Threat analysis summary; remaining risks are stated below and in `docs/security.md` .

Additional mechanisms and rationale. Five further hardening controls are implemented — password hashing, account lockout, audit logging (JSONL, one record per authentication event, never containing a password or message body), session management, and replay protection. Role-based access control was *deliberately omitted*: in a symmetric direct-messaging application any authenticated user may message any roster member — that is the product — so a role model would add attack surface without serving the application. In addition: every remotely controlled string the terminal client prints is sanitized so control sequences render as visible escapes rather than driving the operator's terminal; the dashboard publishes through a hard field allowlist, binds loopback only, and validates the HTTP `Host` header against DNS rebinding; and the test-only fault injector re-checks after `bind()` that it is actually on loopback before arming.

Remaining risks, stated rather than hidden: the server reads plaintext after TLS (transport encryption, not end-to-end); there is no per-client rate limiting, so an authenticated client can flood; in-memory state resets on restart, so a pre-restart message ID could in principle be reused; and lockout can be triggered deliberately to deny a known user service for its window. A user who accepts the wrong self-signed certificate defeats transport authentication — fingerprint comparison through a trusted channel is procedure, not code.

5 Reliability and Fault Handling

Four reliability mechanisms are implemented; each is listed with the problem it addresses, how it works, and how it was tested.

- **Heartbeats.** *Problem:* a silently dead peer (cable pulled, VM paused) otherwise looks connected forever. *How:* after authentication the server probes each session (`PING` every 15 s, 10 s answer deadline); a miss produces a correlated fatal `HEARTBEAT_TIMEOUT` and eviction through the ordinary disconnect path. *Tested:* unit and integration tests on the scheduler and timeout path; exercised live in the failure demo.
- **Acknowledgement ledger.** *Problem:* a sender otherwise cannot distinguish "delivered", "read", and "lost". *How:* every DM has a monotonic outcome — `ACCEPTED → DELIVERED → READ` , or `FAILED` with a stable code — reported under the original message ID and rendered as visible labels in both clients. *Tested:* router unit tests, TCP/WS integration tests, and a Playwright test driving the chain cross-transport to `READ` .

- **Bounded reconnection.** *Problem:* clients must survive a server restart without hammering a dead or flapping host. *How:* both clients share one policy — delays 0.5/1/2/4/8/10/10/10 s with ±20 % jitter, hard stop after eight attempts, manual reconnect after that. The budget refreshes only after 30 s continuously online, so a flapping server is not retried forever; fatal outcomes (bad credentials, version mismatch) are never retried. *Tested:* a nine-socket unit test pins the budget semantics; the browser E2E suite exercises real reconnection.
- **Bounded writes.** *Problem:* one stalled TCP peer could otherwise block roster broadcasts and heartbeats for everyone. *How:* every accepted socket carries a kernel-level `SO_SNDTIMEO` deadline (10 s); an expired write is fatal for that connection only. *Tested:* platform-split unit tests (the mechanism differs on Windows vs POSIX); CI runs both arms.

One decision governs failure handling end to end: **an indeterminate message is never resent automatically.** When a connection dies, in-flight DMs become locally `UNKNOWN`; an explicit retry (`.retry` in the terminal, a button in the browser) always allocates a new message ID. Automatic-resend duplicates are impossible by construction, and the recorded failure run (§7) measured the cost of that honesty: failure detected in 0.05 s, reconnection as a new identity in 0.88 s, and exactly one delivery of the explicitly retried message.

6 Major Challenges Encountered

Technical: the test harness silently lost one AUTH frame in twenty

What happened. Integration tests began failing rarely and unreproducibly — roughly one run in twenty lost a single `AUTH` frame between a test client and the server. *Why.* Escalating instrumentation showed the harness was using one `ssl.SSLSocket` from two threads while socket timeouts were enabled — undefined behaviour in CPython's `ssl` binding that corrupts the record stream just often enough to be maddening. The production code was innocent. *Resolution.* All harness SSL calls were confined to one thread, and the production boundary was audited to confirm it never shares an SSL object across threads — now a documented invariant in the security notes. *Learned.* "Flaky" is a diagnosis of last resort: this was deterministic undefined behaviour with a low trigger probability, and rerunning until green would have shipped a harness able to mask real protocol regressions.

Technical: the browser retried a flapping server forever

What happened. A review probe against a deliberately flapping server showed the browser client stuck in an infinite loop — 21 consecutive generations all logging "attempt 1/8". *Why.* The reconnect budget refreshed on every successful authentication, so a server that accepts, authenticates, then immediately drops clients kept resetting the counter; the policy matched the terminal client's in constants but diverged in semantics. *Resolution.* The budget now refreshes only after 30 seconds continuously online; the rule is pinned on both clients — a nine-socket test on the browser, a policy unit test on the terminal. *Learned.* Two implementations of "the same" policy drift where the specification is silent — the constant table was never the policy; the refresh rule was.

Design: what should a retry mean?

The hardest design argument was retry semantics after a delivery failure. Automatic retransmission (at-least-once) looks helpful but can deliver duplicates — unacceptable when the product's core promise is a truthful per-message outcome. Exactly-once over an unreliable network is not achievable without end-to-end deduplication state the bounded in-memory design deliberately avoids. Relay commits to **at-most-once with honest uncertainty**: indeterminate messages are marked `UNKNOWN`, never silently resent, and an explicit retry is a *new* message with a new ID. This traded a little convenience for an invariant that could be stated, tested, and demonstrated — and it simplified every failure path downstream, because no component ever needs to ask "has this been sent before?"

Process: tests that cannot fail

The recurring failure mode was not wrong code but tests incapable of catching it: environment-gated skips quietly removing coverage, a fault-injection test that passed even against a faked failure path, and — the starkest case — the system's headline invariant ("a retry always allocates a new ID") having *no test at all* for five phases while every review pass assumed it was covered. The countermeasures adopted were mechanical: CI fails the build on a **single skipped test**, and a test is not trusted until it has been seen to fail against deliberately broken code (mutation discipline). Both caught further defects after adoption (§7).

Process: one developer and AI assistance

Relay was built end to end by a single developer, using generative AI heavily as a force multiplier. The scale of the work was made tractable with an orchestration harness and purpose-built review loops and workflows, which made quick work of complex tasks that would normally fall to a team. All multi-host demonstration work was carried out by a single human operator on the author's own hardware — the physical Windows host driving a bridged VirtualBox Ubuntu VM as the second host. Review passes were run after every major phase or implementation, capped to force convergence on the goal rather than polishing loops.

7 Testing and Evaluation

Method. The suite is Python `unittest` under a custom runner adding two guarantees a bare run lacks: any test that leaks a running thread fails by name, and skips are a first-class count — CI fails on a single skip, because a skipped test is a test that cannot fail. CI runs the core suite on Linux (Python 3.10/3.12/3.13) and Windows, plus a separate job driving real Chromium over real WSS against a real server subprocess (eight Playwright tests). Documentation catalogs (message types, error codes, audit events) are parity-tested against code in both directions. Totals at the release commit: core suite 425 run, 0 skipped, 0 failed, 0 errors; browser E2E 8 run, OK — both exit 0.

Test	Procedure	Expected	Actual
Normal operation	Start server; authenticate three participants on two hosts; exchange cross-transport DMs and replies	Roster converges; outcomes reach `READ`; labels visible in both clients	As expected — pinned continuously by the TCP/TLS/WS integration suites; rehearsed end-to-end on the two-host rig (2026-08-06)
Invalid input	Wrong password ×5; unknown recipient; committed probe tool sends three malformed frames over real TLS	Generic failure + lockout; local refusal; correlated fatal `ERROR` per probe; **no crash**	As expected — server kept serving through all probes; protocol strictness pinned by unit tests
Failure scenario	One-shot injector closes the recipient's transport immediately before the delivery write	Sender gets retryable `FAILED`; dead session evicted; browser reconnects; retry delivers exactly once under a new ID	As expected — detection 0.05 s, reconnection 0.88 s, one delivery; recorded end-to-end with SHA-256-hashed artifacts

Table 3: Acceptance testing across normal operation, invalid input, and an injected failure. The failure scenario is recorded in `docs/failure-injection.md` with hashed transcript and screenshots.

Adversarial results worth reporting. The process caught defects polite testing would have missed: the reconnect-budget and harness bugs of §6; a dashboard column that rendered but was never wired to data; a genuine delivery failure that published no `dm.failed` event; and a login overlay every logical test passed while real CSS kept it on screen — found only when a real browser drove the real server. One honest disclosure stands: in 21 suite runs on Aug 1, exactly one uncaptured run reported `1 errors` and never reproduced across 18 consecutive captured clean runs; it is recorded as unexplained, and the process now captures every totals-bearing run.

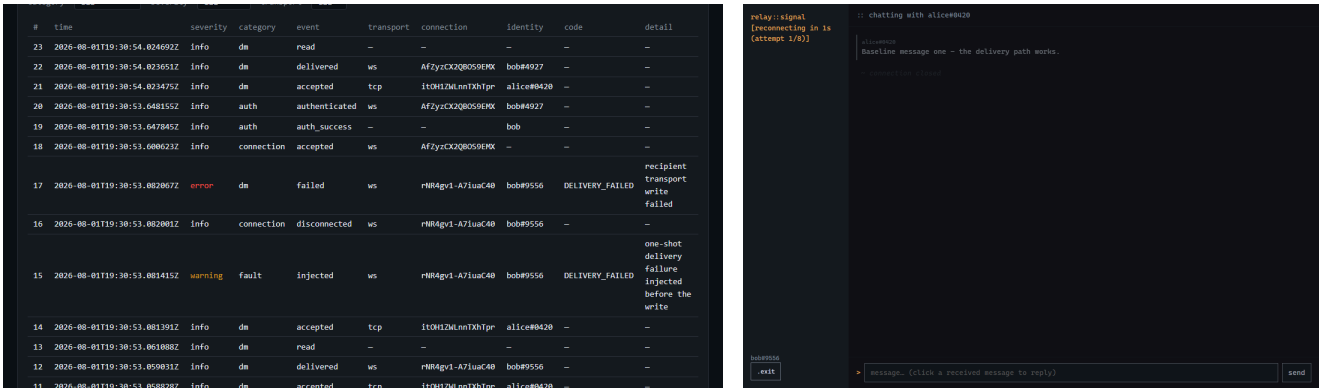

Figure 3: Recorded failure run. Left: excerpt of the dashboard's event feed around the injected delivery failure — the `fault.injected` warning, the `dm.failed` error with code `DELIVERY_FAILED`, and the eviction (full capture in `docs/evidence/`). Right: the browser client reconnecting under bounded backoff, with the failed message's visible outcome and explicit retry control.

The full two-host procedure was rehearsed end-to-end on 2026-08-06 (Host A: physical Windows 11 PC; Host B: bridged VirtualBox Ubuntu VM with its own LAN address) and captured as twelve screenshots (thirteen hashed files — one is a cropped detail of another), cataloged in `docs/evidence/d2/manifest.md`; the rehearsal-record table in the demo runbook holds the observed values. Real LAN addresses are redacted from the published captures.

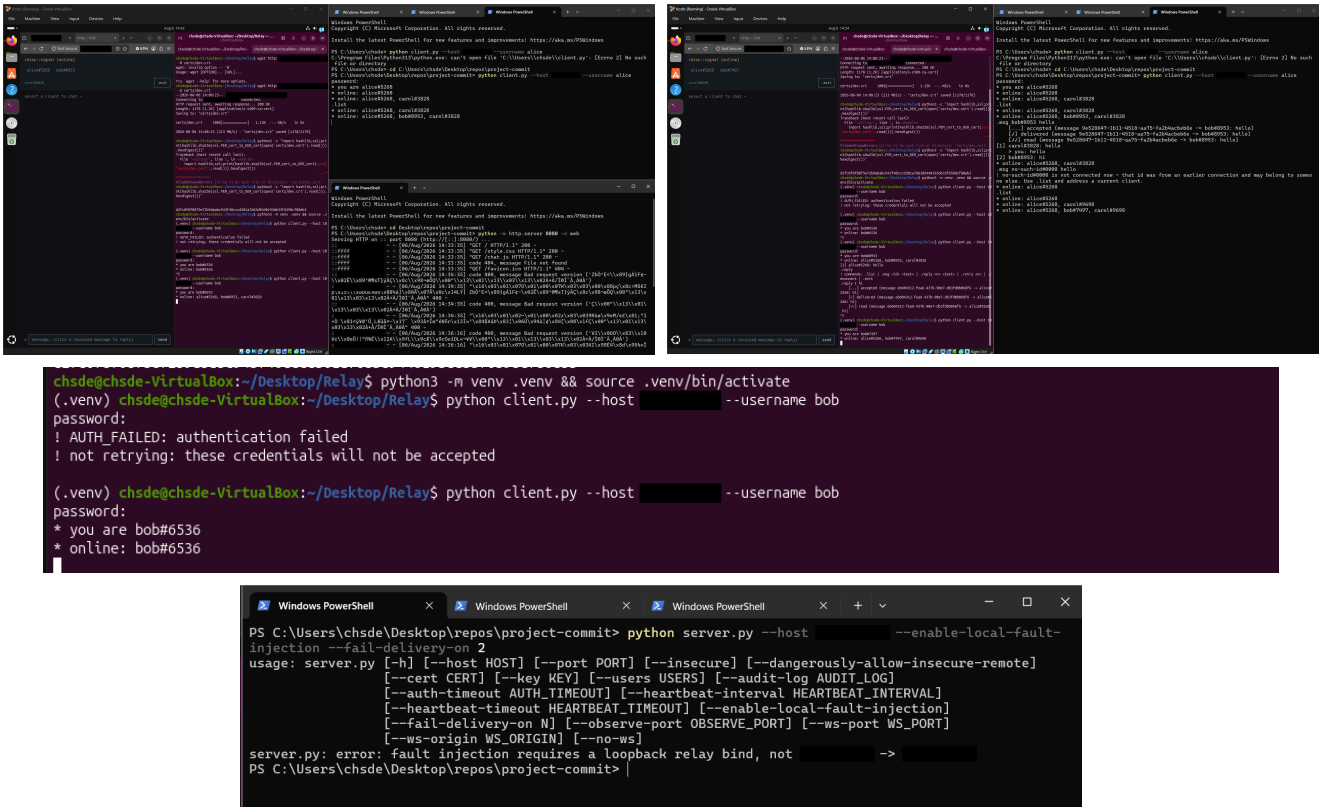

Figure 4: Multi-host rehearsal evidence (addresses redacted). Top left: three simultaneous participants (`alice#5268`, `bob#8953`, `carol#3828`) across both hosts — browser and terminal on the VM, terminal on Host A. Top right: after the VM's network drop — heartbeat eviction shrinks Host A's roster, then reconnection under *new* identities (`carol#3828 → carol#9698`, `bob#8953 → bob#7497`). Middle: wrong password drawing the generic `AUTH_FAILED` with no retry, then success as `bob#6536`. Bottom: the fault injector refusing a LAN bind at argument validation.

Performance evaluation. *What:* application-level send-to-`DELIVERED` latency — one framed DM through TLS decode, validation, routing, bounded retention, the recipient's write, and the outcome write back to the sender. *How:* a committed measurement tool, 200 samples after 20 warmup, over real TLS on loopback on an otherwise idle machine; every CSV row is stamped with commit, platform, and settings. *Results:* median **0.175 ms**, p95 **0.317 ms**, min 0.129 ms,

max 0.401 ms (full CSV and method in `docs/evidence/latency-loopback-tls.md`). *Interpretation:* loopback is the controlled floor, not a network claim — it shows protocol, validation, and routing overhead is sub-millisecond at the median, so deployed latency is dominated by the network path rather than by Relay's message handling. As arithmetic, the median implies a sequential ceiling near 5,700 messages/second on this path; the same tool accepts `--host` for a LAN measurement without code changes.

8 Lessons Learned

What worked well. Making reliability claims *testable* paid for itself: the never-auto-resend invariant, the skip-free rule, and the parity tests between documentation and code each caught real regressions. The two-transport/one-router architecture proved the protocol transport-independent essentially for free, and recording evidence with hashes and commit stamps turned "trust us, it worked" into an auditable artifact.

What did not. Unit coverage kept giving false confidence at integration seams — four separate defects lived *between* proven-correct units — and claims written from memory kept drifting from what the code actually did until the "run it, then quote it" rule was imposed.

What would be redesigned. The browser client would gain stalled-socket detection (it answers heartbeats but never expects them — the one reliability asymmetry left); per-client rate limiting would move from "remaining risk" to implemented; and a persistent replay ledger would survive restarts.

What became clearer. Networking: TCP's reliability guarantees stop at the socket — the application still owns framing, liveness, ordering visibility, and send-buffer stalls, which is precisely what the envelope, heartbeats, and write deadlines exist for. Security: authentication is a *channel* property, not a message property — identity must be bound to the connection and stamped by the server, never asserted by the peer; and the cheapest, highest-value features were those that refuse to run dangerously by default — the fail-closed plaintext gate, the loopback-only injector and dashboard.