Sniffgate: Design and Measurement of a Compact TCP/UDP Failover Gateway

Daozhendong Yang
School of Intelligent Science and Engineering,
Yunnan Technology and Business University, Kunming, China
hi@yxpil.com

Abstract

Sniffgate is a single-process TCP/UDP failover gateway written in Rust: immutable routing snapshots published through `ArcSwap`, active probes over either transport, passive ejection on forwarding failures, hot configuration reload, and deadline-based draining. This paper measures the implementation at commit `1793344` (3,215 lines, a 2.88 MiB binary) on Windows 11 loopback, using a harness that kills real backends and records every latency in a JSON artifact. Under an identical fail-stop, detection is 259.69 ms with UDP probes and 1,143.67 ms with TCP probes, because Windows drops SYN packets to closed ports but resets UDP sockets immediately. An open-but-silent listener takes 1,138.34 ms, which is the probe timeout rather than the transport. The first request after a kill reaches the standby in 510.73 ms through connect retry, well before the health plane switches at 1,198.72 ms; after restart, traffic returns in 195.24 ms. A grid over the probe interval and the failure threshold shows that detection is a schedule rather than a constant: it is roughly t probe cycles, while the client-visible first request stays between 304.01 ms and 327.19 ms across all 24 configurations. We also report negative results. `probe = none` is implemented as a successful probe, so it clears the passive failure window: six requests each pay about 507.87 ms and no ejection occurs, and with a threshold of one a dead primary is resurrected twice, 57.18 and 24.16 ms after the two ejections. Draining is keyed to the union of TCP and UDP roles, so a TCP-only switch leaves the old connection alive beyond 2,203.69 ms, and a zero drain timeout disables draining. A UDP-only failure is invisible to TCP probes: all 100 datagrams are lost over 3,095.63 ms while the node is healthy. Overhead is 1.749× persistent RTT and 0.031× bulk throughput, with 7.57 MB RSS. Six lessons follow.

Artifact. The gateway source is at commit `1793344` (2026-10-04). The measurement harness, the raw artifact `sniffgate_measure.json` (generated 2026-10-24), the figure script, and the table script are in the paper directory. Every number in this paper, including the tables and figures, is generated from that artifact; none was typed by hand. Section 8 gives the reproduction commands.

1 Introduction

A small service often outgrows a single process before it outgrows a single host. The operator then needs a second backend, a way to send traffic to it when the first fails, and a story for what happens to connections that were already open. The two conventional answers sit at opposite ends of a spectrum. At one end, VRRP (Nadas, 2010) or keepalived (Keepalived Project) moves a virtual address between hosts; this is robust and cheap, but it operates at the network layer, needs privileges, and says nothing about which application port is healthy. At the other end, a software L4 proxy such as HAProxy (HAProxy Technologies), Envoy (Envoy Project), NGINX (NGINX), or a system such as Maglev (Eisenbud et al., 2016) offers health checks, connection draining, and load-balancing policies, at the cost of a substantial configuration surface and, often, a substantial process.

Sniffgate is an attempt at the middle: one small Rust process that owns a set of TCP and UDP listen addresses, routes each to an active backend from a static node list, probes the backends, hot-reloads its TOML configuration, and drains old connections after a switch. It is not a load balancer for throughput and does not attempt kernel bypass, TLS termination, or consensus. The entire implementation is 3,215 lines across 11 files, builds to a 2.88 MiB binary, and has 13 unit and 3 integration tests. Its central design choice is that routing state is an immutable snapshot published through `ArcSwap` (arc-swap Contributors): data-path tasks borrow a snapshot, the reload path builds a new one, and no reader ever takes a lock on the hot path.

This paper asks a narrower question than "is it fast?". It asks what a compact health-checking gateway actually does when backends fail in different ways, and whether its documented semantics survive contact with a real host. The answer is mixed, which is the point. We contribute:

C1. a design in which TCP and UDP share one health model and one snapshot, but have separate active sets and separate data-path state (§2);

C2. a measurement method built around tagged echo backends, calibrated host failure semantics, and a JSON artifact from which all tables and figures are generated (§3);

C3. evidence that detection time is a property of the failure

manifestation *and* the probe transport: 259.69 ms versus 1,143.67 ms under the same fail-stop, and 1,138.34 ms for a gray failure (§4.2);

C4. four negative results: a no-op probe that neutralizes passive ejection and resurrects dead nodes (§4.4); drain that is keyed to the union of protocol roles (§4.5); a UDP failure invisible to TCP probes (§4.6); and a fail-open mode that accepts and then hangs for about a second (§4.5);

C5. a forwarding-cost characterization: 1.749× persistent RTT, 0.031× bulk throughput, 8.4% of one core during a 32 MiB transfer, and 7.57 MB resident memory (§4.7).

Non-goals. Sniffgate does not replicate state, elect a leader, or fail over itself; it is a single point of failure in front of two backends. It does not authenticate its admin port; the default bind is loopback, and we treat the admin plane as a local operator interface. It does not terminate TLS or inspect L7. It is not tested across hosts, on a WAN, or under packet loss. Where these gaps matter, we say so rather than extrapolating.

2 Design

2.1 Process, snapshots, and data path

A configuration file names one or more *listeners*, each with a listen address, a protocol (TCP, UDP, or both), and a list of nodes with per-protocol backend addresses. Figure 1 shows the resulting structure. Every listener owns an accept loop (TCP) (Eddy, 2022) or a receive loop (UDP) (Postel, 1980; Eggert et al., 2017). On the TCP side, an accepted connection selects the active node for its listener, connects to the backend with a bounded retry loop, and then splices bytes in both directions with Tokio's `copy_bidirectional` (Tokio Project). On the UDP side, the first datagram from a client address creates a session with its own upstream socket; subsequent datagrams from that address reuse it, and the session is rebound to the new active backend after a switch. Sessions expire after 60,000 ms of inactivity.

The routing state is a `Runtime` value: node table, per-protocol active node, drain deadlines, and a monotonically increasing epoch. Readers call a single load on an `ArcSwap`, which yields an `Arc` to the current snapshot; writers build a complete replacement and publish it. An in-flight TCP connection keeps the snapshot it started with for routing decisions, but subscribes to a separate `watch` channel for drain deadlines, so a later switch can still close it at the right time. This is deliberately RCU-like rather than lock-based: the common case is an atomic load and an `Arc` clone, and the uncommon case (reload, health transition) pays for allocating a new value.

2.2 Health: active probes and passive ejection

In the tuning used throughout this paper (Table eftab:config), each node runs a probe loop every 100 ms with a 300 ms timeout; the stock defaults are 1,000 ms and 800 ms. A TCP probe connects, writes an optional payload, and waits for a response containing an expected string; an empty payload and expectation reduce it to a connect check. A UDP probe sends a payload and waits for any response. The `auto` policy uses TCP when a TCP address exists and UDP otherwise. The `none` policy returns success without sending anything; §4.4 shows why this is not the same as disabling active health checks.

Health is a small state machine. Three consecutive probe failures mark a node unhealthy; two consecutive successes mark it healthy again. Independently, every forwarding failure (a failed connect, in the current implementation) is recorded against the node. If 3 such failures occur within 5,000 ms, the node is ejected immediately, without waiting for the probe loop. The subtle part is that a successful probe *clears the passive failure window*. That is defensible when a probe exercises the same path as data, but it couples two signals that should be independent, and it is the root cause of the behavior in §4.4.

Selection is by priority, round-robin, or weighted choice among healthy nodes. Priority is the default: the first healthy node in configuration order is active, and the next healthy node becomes active when it is demoted. Connect retry is separate from health: a client connection may fail against the active backend and be retried against the standby before the health plane has noticed anything. This distinction explains why client-visible recovery can be much faster than detection.

2.3 Draining and UDP sessions

When a node loses its active role, existing TCP connections should not be cut immediately: a client may be mid-request. Sniffgate implements draining with an absolute deadline of the configured length (1,000 ms in our experiments, against a stock default of 10,000 ms). The publisher computes the set of nodes that are active for at least one protocol, and every node that is absent from that set and has a positive drain timeout starts a drain deadline at $now + 1,000$ ms. Each TCP connection watches the deadline; at expiry it stops copying and closes. New connections route to the new active node immediately. A zero timeout skips draining altogether.

The scope of that set is the design decision worth flagging: it is the *union* over TCP and UDP. A node that is demoted for TCP but still active for UDP remains in the union, so it is not drained. §4.5 measures the consequence: a TCP-only switch does not drain, and the old TCP connection survives until the client closes it.

UDP sessions are keyed by client address. On a switch event for the UDP protocol, each session optionally rebinds its upstream socket to the new active backend; the client address and its mapping do not change. There is no retransmission queue and no application-level acknowledgment, so any datagram in flight during the rebind is at the mercy of the host's socket buffers. On loopback, §4.6 finds no loss.

2.4 Configuration and admin plane

The configuration is TOML. A watcher polls the file's modification time and size every 100 ms in our experiments (the stock

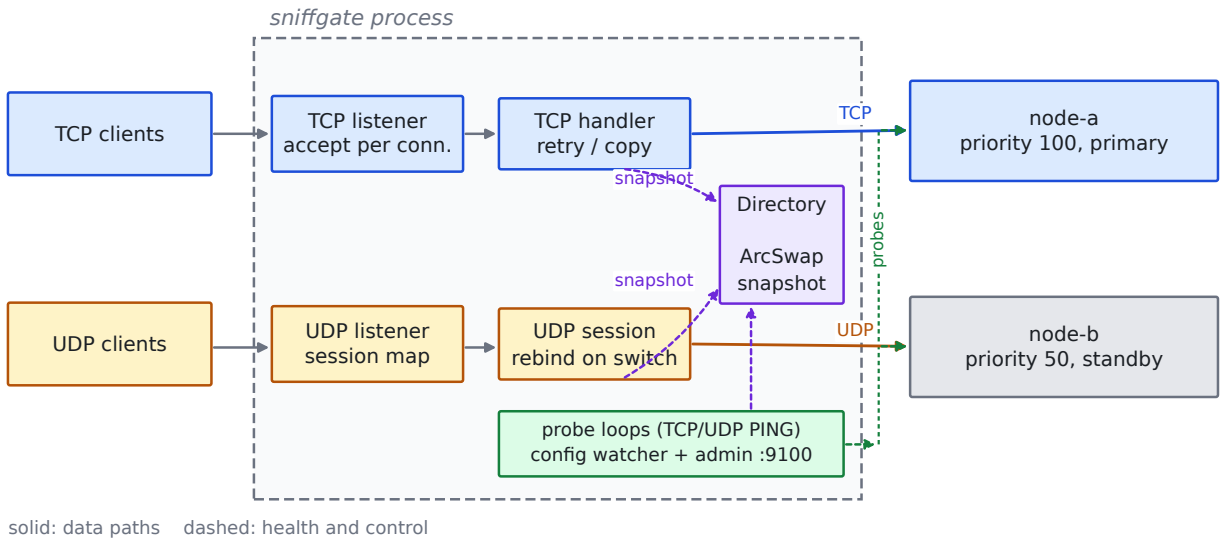


Figure 1: Sniffgate's data and control paths. TCP and UDP listeners share one immutable routing snapshot. Probe loops publish health transitions; the config watcher rebuilds the snapshot on a valid reload; the admin port exposes status and commands. Data-path tasks read the snapshot without a lock.

default is 1,000 ms), parses and validates a candidate, and publishes a new snapshot only if validation succeeds. A failed parse leaves the old snapshot and epoch in place, which makes a bad edit a no-op rather than an outage. The admin plane is a line-oriented TCP protocol on `127.0.0.1:9100` by default: one command per line, one JSON object per response. It supports status, stats, reload, manual switch, automatic mode, health overrides, and per-node enable/disable. There is no authentication and no line-length cap. The loopback default is the only access control, which is adequate for a local operator tool and inadequate for anything else; we return to this in §4.1 and §7.

3 Measurement Method

3.1 Harness and artifact

The harness is a Python 3.13 program using only the standard library. Every scenario starts a fresh gateway process and fresh backend processes on loopback, so no health state, connection, or counter leaks between trials. Backends are tagged echo servers: a TCP request receives `ID <node>` and a UDP request receives the same tag, which lets the harness attribute every response to the backend that produced it. Failures are injected by terminating the backend process (fail-stop), by replacing it with a listener that accepts connections and never answers (gray failure), by closing only its UDP socket, or by switching traffic with the admin command. The three failure modes exercise different parts of the health model.

Client latency is measured with `perf_counter` after requesting a 1 ms timer resolution. Gateway CPU time and resident memory are read through the Win32 process APIs. Control-plane timestamps are taken from the gateway's own status response and cross-checked against the harness clock on the same host; the paper reports both where they differ in definition. The harness writes one JSON object per scenario, including raw trials, percentiles, and outcomes. The tables in this paper are generated from that JSON by `scripts/make_tables.py`, and the figures by `scripts/make_figures.py`. Table 1 lists the build and the default tuning used unless a scenario overrides a value.

Table 1: Build facts and the experimental tuning. The gateway's stock defaults are slower (1,000 ms probe interval, 800 ms probe timeout, 3,000 ms connect timeout, 10,000 ms drain, 1,000 ms config poll); we shorten them so that failures and reloads complete within bounded time. All comparisons use the values below unless a scenario says otherwise.

Build and host	
Repository commit	`1793344` (2026-10-04)
Compiler	rustc 1.98.1
Binary	3,015,168 B (2.88 MiB)
Source	3,215 lines in 11 files
Tests	13 unit + 3 integration (537 lines)
Host	Windows 11, Intel Core Ultra 5 125H
Experimental tuning	
Probe interval	100 ms
Probe timeout	300 ms
Failure / success threshold	3 / 2
Connect timeout / retries	500 ms / 3
Passive ejection	3 failures in 5,000 ms
Drain timeout	1,000 ms
Config poll	100 ms
UDP session idle	60,000 ms
Failure policy / strategy	fail-open / priority

Each scenario has a small number of trials: one for startup, reload, and the failure-mode probe; two to five for failover; six client attempts for passive ejection. We report medians

Table 2: How a dead backend looks to the transport layer on the measurement host (loopback, Windows, three probes each). A TCP connect to a port with no listener is silently dropped, so the failure costs one probe timeout before it is observable; a UDP send to a closed socket is refused immediately, so it costs one syscall. This single asymmetry explains the two halves of Table 5.

Transport	Failure signal	Median
TCP	TimeoutError, delayed	200.4 ms
UDP	WinError 10054, immediate	0.05 ms

and full ranges rather than confidence intervals, because the sample sizes do not support inference. A number without a range is a single observation or a directly measured aggregate.

3.2 Host failure semantics

Loopback is not a neutral medium. Two measured properties shape every result. First, a TCP connect to a closed port on this Windows host does not receive a reset: the SYN is silently dropped, and the caller waits for its own timeout. A 200 ms client timeout expired after 199.52–200.97 ms across three probes; no RST arrived. The gateway's TCP probe uses its 300 ms timeout, so a probe against a dead port costs the timeout, not a round trip. Second, a UDP send to a closed port returns `WSAECONNRESET` (10054) (Microsoft) in 0.04–0.16 ms, driven by the ICMP port-unreachable that Windows delivers to the socket. A UDP probe therefore fails almost instantly.

This asymmetry is the single most important confounder in the paper, and it is not a Sniffgate property. It means that the same fail-stop is detected at different speeds depending on which transport the probe uses. We did not average the two or report only the faster one; both arms are measured under otherwise identical conditions (§4.2). Table 2 collects the two measurements.

A third host property affects the connection-setup numbers: a fresh loopback TCP connection is bimodal, clustering around 0.52 ms at the median and 10.88 ms at the 95th percentile even when connecting directly to a backend. The 10 ms mode is a scheduler artifact of the host, not the gateway. We therefore compare direct and proxied connections in the same session and treat the p95 difference, not the tail itself, as the signal.

3.3 Threats to validity

The experiments run on one Windows machine over loopback, with echo backends, one gateway binary, and no TLS, no WAN, no packet loss, and no multi-host routing. The measured detection times depend on Windows socket semantics; a Linux host would likely reset a closed TCP port immediately and change the TCP-probe arm by nearly an order of magnitude. The throughput numbers measure one core and one memcpy path, not a production forwarding capacity. The sample sizes are small, the backend work is trivial, and the client is a Python process. We treat the results as a characterization of this implementation and this host, and we mark every place where the mechanism, rather than the exact number, is the transferable claim. Table 1 gives the commit and binary hash so that the same binary can be re-measured elsewhere.

4 Results

4.1 Startup, control plane, and reload

A fresh process opens its admin socket 72.13 ms after spawn and its data plane 90.34 ms after spawn. The difference is the time to bind listeners and publish the first snapshot; both include process startup, since the harness timestamps the spawn. Once up, a `status` call costs 82.60 μs at the median and 198.10 μs at p95 over 200 calls, with a maximum of 351.20 μs. The control plane is not on the data path, but an operator polling it during an incident should know that it does not block forwarding.

A new admin connection costs 0.54 ms at p50 and 10.58 ms at p95 over 20 connections; the p95 is the same 10 ms host mode seen below. Establishing a data connection through the gateway costs 0.49 ms at p50 (10.52 ms at p95), against 0.52 ms (10.88 ms) to the backend directly over 60 attempts. The median difference is within the host's own jitter, so this experiment cannot resolve a setup cost; it does show that the gateway does not add a fixed millisecond-scale delay.

Idle probing is cheap. Over 3 windows of 4 s each, each node received 9.00–9.25 probes per second, and the gateway's CPU share ranged from 1.172% to 6.250% with a mean of 3.26%. The 1 ms timer quantizes a 4 s window into 0.025% steps, so the zero reading is a floor, not evidence of no work. Resident memory after startup is 7.12 MB.

Reload converges in roughly a quarter of a second with the shortened 100 ms poll interval; with the stock 1,000 ms poll it would be bounded by that poll instead. Adding a third node was observed at 249.53 ms from the admin side, 242.77 ms on the gateway's internal clock, and 260.55 ms at the data plane, with epoch 3 and traffic moving to the new node. A deliberately invalid TOML file was detected at 250.37 ms (internal 245.68 ms), was rejected, and left both the epoch (3) and the active node unchanged, while the data plane continued to serve the previous node. Removing the node took 259.05 ms (254.29 ms internal, 260.04 ms data), advanced the epoch to 4, and moved traffic back. An explicit `reload` command costs 0.16 ms at p50 and 0.35 ms at p95 over 20 calls, but that number only measures the command acknowledgment; the observable convergence is the polling interval plus parse and publish, which is the 242.77–254.29 ms internal range above. Table 3 collects the control-plane numbers.

The admin parser answered all 4 hostile inputs with `ok: false`: garbage (113 response bytes in 9.89 ms), an injection-shaped line (125 bytes in 11.64 ms), a switch to a nonexistent node (72 bytes in 10.63 ms), and a 65,536-byte line. The last one produced a 65,645-byte response in 12.56 ms: the parser reads an unbounded line and echoes it in the error message.

Table 3: Control-plane latency. Percentiles are shown as p50 (p95). Reload rows report admin-observed / gateway-internal / data-plane observation in that order.

Metric	p50 (p95)	n
Spawn to admin socket	72.13 ms	1
Spawn to data plane	90.34 ms	1
`status` RPC	82.60 (198.10) μs	200
New admin connection	0.54 (10.58) ms	20
Connect to backend directly	0.52 (10.88) ms	60
Connect through data port	0.49 (10.52) ms	60
Connect through admin port	0.58 (10.91) ms	60
Reload: add node	249.53 / 242.77 / 260.55 ms	1
Reload: invalid file	250.37 / 245.68 ms	1
Reload: remove node	259.05 / 254.29 / 260.04 ms	1
Explicit reload	0.16 (0.35) ms	20
Resident memory	7.12 MB	1

The data plane was intact after all four. This is parser robustness, not security. With the default loopback bind the exposure is local; an operator who moves the admin port to another interface would need authentication and a cap.

4.2 Detection depends on the failure

Table 4 and Figure 2 summarize the main failover experiment. In five active-failover trials, terminating the active backend produced an internal switch at 1,198.72 ms (range 1,156.19–1,220.92), observed by the admin at 1,204.08 ms. The reason is the probe cost: a TCP connect to a dead loopback port waits for the 300 ms probe timeout, then the loop sleeps 100 ms; three failures therefore cost about 1.2 s. The switch record shows three probe failures and zero passive ejections, so the health plane, not a client failure, caused the switch.

The client does not necessarily wait for that switch. The first request issued immediately after the kill succeeded on the standby in 510.73 ms (range 506.77–527.66). The gateway's connect retry tried the dead active node, exhausted one connect attempt, and retried the standby; the request-level path was faster than the health plane by more than a factor of two. Every request in every trial reached the standby, and every verification request after restart reached the primary. This is good for availability and bad for a simple story: the control-plane detection time is not the user-visible outage. We report both, and we suggest that any gateway paper report both.

Recovery after restarting the backend took 195.24 ms at the admin (range 188.99–228.76) and 191.45 ms internally. Two successful probes are required, so recovery is at least one interval plus one round trip; the measured value is consistent with that floor and adds little. The contrast with detection is the useful part: recovery is fast because the probe succeeds immediately, while detection is slow because the probe must time out three times.

A gray failure—an open listener that accepts and never answers—switched at 1,138.34 ms (range 1,134.13–1,142.55), observed at 1,142.18 ms, with 3 probe failures and recovery at 199.34 ms. The number is almost identical to the TCP fail-stop arm, which is expected: the connect succeeds, the PONG read waits the same 300 ms, and the state machine sees the same three failures. Sniffgate cannot distinguish "the process is gone" from "the process is alive but not serving"; both are a probe timeout. That is acceptable for a binary health model, but it means the gateway learns nothing about the gray failure beyond its own patience.

The controlled comparison is the top two rows of Table 4: same fail-stop, same three trials, only the probe protocol differs. UDP probes detected at 259.69 ms (range 250.75–296.73); TCP probes at 1,143.67 ms (range 1,111.57–1,154.40). The gap is the host asymmetry from §3, amplified by the failure threshold: a UDP probe fails instantly and three intervals dominate, while a TCP probe pays a timeout and an interval for each failure. Both arms switched all TCP and UDP requests to the standby, both recovered, and neither recorded a passive ejection. The result is not that UDP probing is better; it is that a probe only measures the transport it uses, and on this host the TCP transport reports death slowly. A gateway that probes TCP for a UDP service inherits the wrong clock.

4.3 Detection time is a schedule, not a constant

Two numbers do not generalize: the detection time depends on how the probe schedule is configured. Table 5 measures detection latency over the cross product of probe interval and failure threshold, 24 cells of 3 trials each, with passive ejection disabled so that only the probe loop can cause a switch. The structure is linear in the threshold and linear in the interval, with a different slope per transport.

At a 100 ms interval, raising the threshold from one to three moves TCP detection from 392.60 ms to 1,195.06 ms, a step of about 322.77 ms per unit of t, which is one interval plus one probe timeout. The same change moves UDP detection from 79.79 ms to 253.15 ms, a step of 86.59 ms, which is one interval and no timeout term. Widening the interval to 1 s instead raises the threshold-three medians to 3,738.08 ms for TCP and 2,836.82 ms for UDP. In both transports the detection time is roughly t probe cycles, where a cycle is one interval plus one timeout for TCP and one interval for UDP, minus the part of the current cycle that had already elapsed when the backend died.

The client-visible side of the same grid barely moves. Across all 24 configurations, the first request issued after the kill reached the standby between 304.01 ms and 327.19 ms at the median. Connect retry, not the health plane, decides how long a client waits, and its cost is set by the connect timeout rather than by the probe schedule. Tuning the probe interval changes when the gateway *knows*, not when the client is *served*. Any evaluation that reports only the detection number therefore reports a property of the configuration, not of the users.

4.4 Passive ejection and the no-op probe

Passive ejection is the safety net for failures that active probes miss or notice late. With probes disabled at the configuration level and the passive threshold set to two, the harness kills the

Table 4: Detection, client impact, and recovery by failure manifestation. *Switch* is the gateway's internal timestamp delta from kill to the published switch; *request* is the first request issued after the kill. Recovery is the time from backend restart to traffic returning to the primary. Gray failure is an open listener that never answers.

Trigger	Probe	n	Switch p50	min–max	Request p50	Recovery p50
Fail-stop	UDP probe	3	259.69	250.75–296.73	–	172.92
Fail-stop	TCP probe	3	1,143.67	1,111.57–1,154.40	–	186.74
Fail-stop	TCP probe	5	1,198.72	1,156.19–1,220.92	510.73	195.24
Gray failure	TCP probe	2	1,138.34	1,134.13–1,142.55	–	199.34
No probes	passive	1	1,032.32	–	–	–

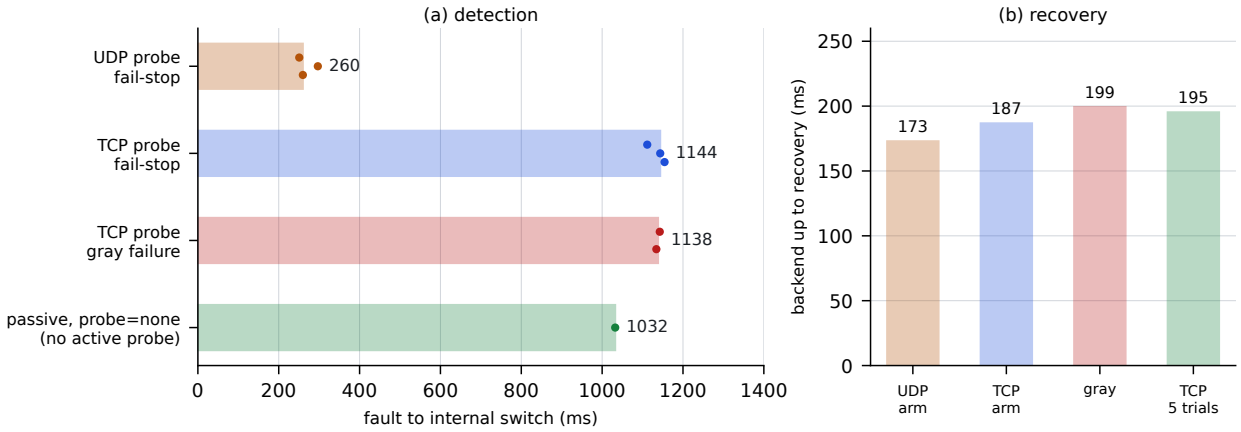


Figure 2: Detection and recovery by failure manifestation. Bars are medians; dots are individual trials where more than one was run. Fail-stop detection differs by more than a factor of four between UDP and TCP probes because of host socket semantics, not gateway logic. Gray failure costs one probe timeout per failed probe. Recovery is similar across arms because it is driven by successful probes after restart.

active backend and then makes six client attempts. Table 6 and Figure 3 show two very different outcomes depending on a single configuration value.

In the clean passive run, attempts 1 and 2 each cost about 517.07 and 515.28 ms, because each first tried the dead primary and then reached the standby by retry. The second forwarding failure reached the threshold and ejected the node; the internal switch timestamp is 1,032.32 ms. Attempts 3–6 cost between 10.11 and 14.10 ms (p50 12.55 ms), all served by the standby, and the dead backend received 0 probe PINGs because there were no active probes to receive them. The final node state was unhealthy. This is the behavior the documentation promises.

Now set `probe = none` and give the probe loop a 100 ms interval, still with passive threshold two. Six attempts cost 505.56–521.90 ms (p50 507.87 ms), *all* of them, and there were 0 passive ejections; the active node stayed healthy and in place. The successful-probe counter rose by 45 over the run. The mechanism is visible in the source: `probe_once` returns `Ok(0)` for `ProbeKind::None`, and `record_probe(true, ...)` clears the passive failure window on every success. A no-op probe therefore runs every 100 ms and erases the client failures faster than two of them can accumulate. The node is certified healthy by doing nothing, and every request pays the connect timeout against the dead backend before retrying the standby.

The same mechanism can resurrect a node that has already been ejected. With `probe = none`, a passive threshold of one, and a success threshold of one, each client failure ejects the primary and each following no-op probe promotes it again. The first request cost 503.75 ms, the second 504.20 ms. The active-node timeline shows node-a at 1.16 ms, node-b at 506.49 ms, node-a again at 563.67 ms (a resurrection 57.18 ms after the ejection), node-b at 2,170.06 ms, and node-a again at 2,194.22 ms (24.16 ms after the second ejection). The run recorded 2 ejections and 2 resurrections, with the successful-probe counter rising by 23 and the dead backend receiving 0 PINGs. In a deployment with a passive threshold of one, this is a flapping loop that alternates between a 500 ms penalty and a false recovery, forever.

The lesson is not that passive ejection is broken. The lesson is that "no active probe" must be a distinct state, not a probe that succeeds. A no-op probe should not touch the passive window, should not increment a success counter, and should not participate in promotion. The implementation currently conflates "I did not check" with "I checked and it was fine", and the cost is both a disabled safety net and a possible flapping loop.

4.5 Draining is protocol-scoped

Draining is the part of failover that users notice most: a switch that cuts an in-flight request is a visible error, while a switch

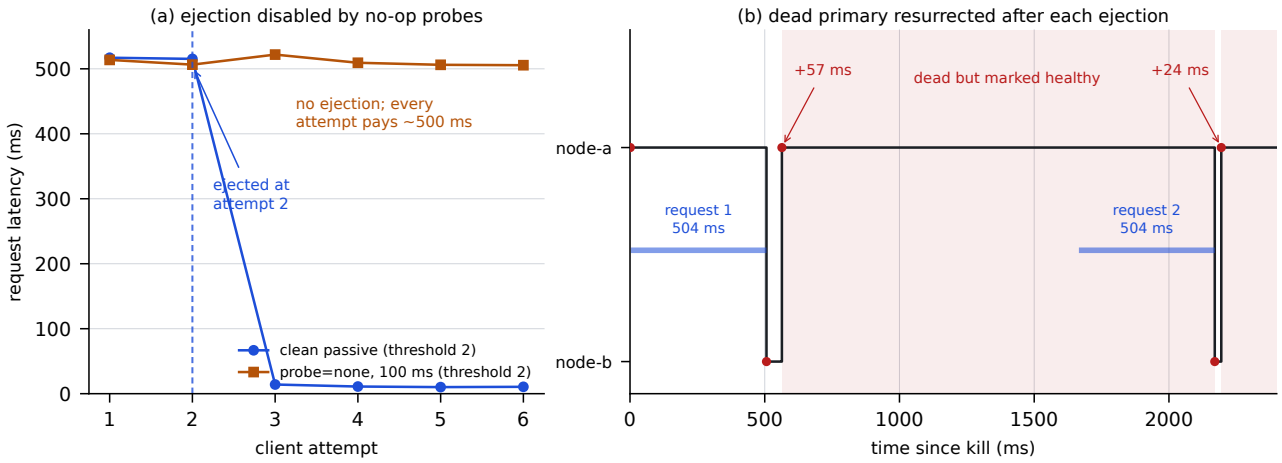


Figure 3: Passive ejection with and without a no-op probe. (a) Threshold two: the clean run ejects at attempt 2 and subsequent requests cost about 10.11 ms; `probe = none` clears the failure window every 100 ms, so no ejection happens and every attempt pays about 507.87 ms. (b) Threshold one: each ejection is followed by a no-op success that promotes the dead node again, producing two resurrections.

Table 5: Active-probe detection latency (median of 3 trials per cell, ms) as a function of the probe interval and the failure threshold, with passive ejection effectively disabled (t set to 10^5) and no client traffic. TCP probes grow as t times (interval plus timeout); UDP probes grow as t times the interval, because a failed datagram is refused at once. Both follow the transport asymmetry of Table 2.

	TCP probes		
Interval	$t = 1$	$t = 2$	$t = 3$
100 ms	392.6	715.4	1,195.1
250 ms	418.8	955.9	1,488.3
500 ms	622.8	1,471.4	2,268.2
1000 ms	1,186.8	2,471.1	3,738.1
	UDP probes		
Interval	$t = 1$	$t = 2$	$t = 3$
100 ms	79.8	166.4	253.2
250 ms	110.2	338.5	631.5
500 ms	340.6	822.9	1,389.4
1000 ms	840.4	1,836.2	2,836.8

Table 6: Passive ejection and the effect of `probe = none`. *Ejects* counts passive ejections; *probe* Δ is the rise in successful probe count over the run, which shows that no-op probes keep certifying a dead backend. The clean run's ejected-at-attempt is 2.

Run	n	p50	min–max	Ejects	probe Δ	Final
Passive, threshold 2	6	12.55	10.11–517.07	1	–	node-b
Probe none, 100 ms	6	507.87	505.56–521.90	0	+45	node-a
Resurrection, 2 cycles	2	503.98	503.75–504.20	2	+23	node-a

that lets it finish is invisible. Sniffgate starts a 1,000 ms drain when a node leaves the active set. The active set is the union of the TCP and UDP principals, so the scope of a switch determines whether a drain happens at all. Table 7 reports three explicit-switch experiments.

When both protocols switch, the old TCP connection is still served by node-a at 700 ms, node-a is marked draining, and the connection closes 1,011.13 ms after the switch (range 1,006.61–1,013.24), which is the 1,000 ms drain plus scheduling. New connections go to node-b. All 3 trials closed within 1.5 s. This is the intended path.

When only TCP switches, node-a stops receiving new TCP connections but remains the UDP principal. Because the union still contains node-a, it is not draining: the old TCP connection is still answered by node-a at 700 ms, has no EOF within 1.5 s, and survives beyond 2,203.69 ms while new connections go to node-b. Zero of 3 trials closed. An operator who switches one protocol on a dual-protocol listener gets no drain at all, even though the TCP role moved. The fix is not subtle—drain should be per protocol, or a node should be drained for a protocol when that protocol's role is lost—but the union is easy to write and hard to notice until an old connection outlives the failover.

A zero drain timeout is documented as an immediate disconnect. In the implementation, the publisher only starts a drain when the timeout is greater than zero; zero means no drain, not a zero-length one. With both protocols switched and `drain_timeout_ms = 0`, the old connection was still alive after 1.5 s and was still served by node-a, while the active node was node-b and the draining flag was false. The documentation and the code disagree, and the measurement follows the code.

The failure mode when all backends are dead completes the picture (Table 8). With fail-closed, the gateway waits 701.71 ms for both probes to fail; the client connection is established after 9.96 ms and reaches EOF 0.13 ms later with zero bytes. With fail-open, it waits 689.03 ms, accepts the connec-

Table 7: What happens to an in-flight TCP connection across a switch ($t_{\text{drain}} = 1000$ ms where set). "both" demotes the node from the TCP and the UDP role, which is what the drain logic keys on; "TCP only" leaves it as the UDP primary, so it never enters the draining set and the old connection is not closed at all. "$t_{\text{drain}} = 0$" is documented as "disconnect immediately" but in the code it disables draining.

Switch	**drain**	**old conn (ms)**	**new conn**	**draining**	**trials**
both roles	on	1,011.1 (max 1,013.2)	standby at once	yes	3
TCP role only	on	2,213.3+	standby at once	no	3
both roles	0 ms	1,500+	standby at once	no	1

Table 8: Behavior when both backends are dead and unhealthy. Times are milliseconds; both rows wait for both probes to fail (701.71 and 689.03 ms respectively).

Policy	Active	Setup (ms)	EOF (ms)	Bytes
Fail-closed (`false`)	none	9.96	0.13	0
Fail-open (`true`)	node-a (unhealthy)	9.81	1,019.21	0

tion in 9.81 ms, keeps the unhealthy node as active, and closes after 1,019.21 ms with zero bytes. Fail-open here does not provide availability: the client neither receives data nor receives an immediate refusal; it waits about a second for a connection that was doomed from the start. If the goal of fail-open is to keep a listener reachable, this implementation meets the letter and misses the intent.

4.6 UDP is a separate failure domain

TCP probes and TCP data share a transport; UDP data does not. Table 9 isolates the consequence. With a TCP probe configured and the UDP backend dead, the harness sent 100 datagrams over 3,095.63 ms: 100 were lost. The gateway reported node-a as healthy and active for UDP, with 30 successful TCP probes during the window. A manual switch took 0.30 ms from the admin, after which the first reply arrived in 0.67 ms. The failure was invisible until an operator intervened, because the only signal the health model had was a TCP probe that did not use the broken path.

The switch path itself is good. During a 810.50 ms burst, the harness sent 200 datagrams and received 200, with 0 lost. The tags show 81 replies from node-a and 119 from node-b, so the session moved mid-burst. The admin switch took 0.24 ms, and the first node-b reply arrived 2.46 ms after the command. On loopback, rebinding a per-client upstream socket mid-burst did not drop a datagram. This is a positive result for the session design and a reminder that the weak point is detection, not the switch.

The negative result is structural: a health model that probes only one protocol cannot see a failure in the other. `probe = auto` selects TCP when a TCP address exists, which is a reasonable default for a dual-protocol service only if both transports fail together. They need not: a process can keep its TCP listener while its UDP socket is closed, or the reverse. The measured 3,095.63 ms of complete UDP loss while the control plane reported healthy is the cost of that assumption.

A second experiment isolates the same blindness from the loss side. Table 10 keeps the TCP backend alive, closes only the UDP socket of the same node, and drives UDP traffic at 200 Hz per socket from 4 sockets for 6 s. In every cell the TCP probe kept succeeding: 0 probe failures were recorded, so the node stayed healthy, 0 passive ejections fired, the passive failure counter ended at 0, and node-a was still reported as the UDP primary. The datagrams lost, 20 of 156, are the ones in flight when the socket was closed; every later datagram was answered by the same dead-backend node, which the gateway considered healthy throughout. Varying the probe interval over 100, 500, and 1,000 ms and the send rate over 20 and 200 Hz per socket does not change the outcome, because the probe never touches the broken path. A passive ejection could still have saved the clients, but it cannot fire: the successful TCP probe clears the passive window on every tick, so the failed UDP sends never accumulate.

4.7 Forwarding overhead

Table 11 and Figure 4 report the cost of the extra userspace hop. Persistent TCP request–reply latency on loopback rose from 38.30 μs at p50 to 67.00 μs through the gateway (1.749×); the p95 rose from 112.20 to 171.70 μs. UDP shows the same shape: 36.10 to 66.10 μs at p50 (1.831×), and 75.00 to 114.50 μs at p95. The UDP proxy uses a per-session task and a second socket, so the similarity to TCP is not accidental: both paths pay one copy in each direction and one scheduler wake-up on each side.

For new TCP connections, the median rose from 1.34 ms to 10.18 ms (7.624×), while the p95 moved from 11.24 to 15.66 ms. Both ends of that distribution are dominated by the host's bimodal 10 ms loopback mode: the direct connect sits on the fast mode and the proxied one on the slow mode, so the ratio mostly records which side of the host's scheduler artifact a connection lands on. The experiment therefore does not resolve the setup cost of the proxy path. What it does show is that the gateway adds no fixed millisecond-scale penalty of its own, because the p50 persistent path—a connection that is already open—costs only the extra copy and wake-up measured above.

Bulk transfer is where the design gives up the most. A 32 MiB echo stream achieved 802.86 MiB/s directly and 24.93 MiB/s through the gateway, a ratio of 0.031. The gateway consumed 1.28 s of CPU, 8.4% of one core, or 292.5 MiB per CPU-second. The data path is `copy_bidirectional`

Table 9: UDP across a switch. A session is keyed by client address and is rebound to the new primary when the switch event arrives; the gateway keeps its own backend socket per session, so in-flight datagrams that were sent to the old backend are lost, while the session itself survives. The lower block is the blind-spot probe of Table 10 measured by the main harness.

Measurement	Value	Measurement	Value
datagrams sent during a switch	200	unanswered after the switch	200
lost across the switch	0	switch → first reply from standby	2.46 ms
replies from old primary	81	replies from new primary	119
blind spot: datagrams lost	100 / 100	over window	3,095.6 ms
blind spot: node health	h	ealthy for 30 consecutive probes	

Table 10: Cross-protocol blindness. TCP probes stay green while the UDP backend of the same node is dead: every successful probe wipes the passive window, so passive ejection never fires and the node stays the UDP primary. The lost datagrams are exactly those in flight when the UDP socket was closed. i = probe interval, r = send rate per client socket, lost/sent = UDP datagrams unanswered in the 6 s window, ok/pf = probe successes/failures counted on the node, UDP primary = the node reported as carrying UDP traffic at the end of the window.

i (ms)	r (Hz)	eject	lost/sent	ok	pf	UDP primary
100	200	0/2	20/156	63	0	node-a, node-a
500	200	0/2	20/156	14	0	node-a, node-a
1000	200	0/2	20/156	7	0	node-a, node-a
1000	20	0/2	20/60	7	0	node-a, node-a

over Tokio, with no zero-copy, batching, or kernel bypass; a single core becomes the bottleneck well before loopback does. For request–reply traffic the cost is a latency multiple, but for bulk traffic it is a throughput fraction, and an operator should size accordingly.

Concurrency is modest but clean: 100 of 100 simultaneous connections succeeded, wall time 75.39 ms, per-connection latency 32.63 ms at p50 and 49.35 ms at p95. Resident memory at the end of the overhead run was 7.57 MB, so the process does not accumulate per-connection state at a scale that would matter for this workload. None of these numbers supports a claim of high-performance forwarding; they support a claim that the implementation is small and predictable.

5 Lessons

Six lessons follow from the measurements, in decreasing order of how much they generalize.

L1. Probe the transport you forward. The same fail-stop was detected at 259.69 ms with UDP probes and 1,143.67 ms with TCP probes. The difference was not in the gateway's state machine; it was in what the host does with a SYN to a closed port versus a datagram to a closed port. A gateway that probes TCP for a UDP service inherits TCP's failure semantics, and the measured 3,095.63 ms UDP blackout with a healthy TCP probe is the extreme case. Per-protocol probe selection is not a convenience; it is a correctness requirement.

L2. "No probe" is not a successful probe. The `none` policy returned success, cleared the passive window, and incremented the success counter. That turned a passive safety net into a no-op, produced six consecutive half-second requests with zero ejections, and, at threshold one, resurrected a dead primary twice within 57.18 and 24.16 ms. A disabled probe should be a third state: no success, no failure, no clearing, no promotion.

L3. Drain scope must follow the role that moved. Draining keyed to the union of TCP and UDP active sets means a TCP-only switch never drains the TCP connection, because the node is still active for UDP. The old connection survived beyond 2,203.69 ms while new connections used the standby. Union semantics is a plausible one-line implementation and a surprising operational behavior; drain should be tracked per protocol, and a zero timeout should mean what the documentation says.

L4. Active probing sees liveness, not correctness. Gray failure and fail-stop produced nearly the same switch time, because both reduce to a probe timeout. That is fine for a binary model, but it means the detection time is set by patience (300 ms times the failure threshold in our tuning), not by the nature of the fault. A shorter timeout detects gray failures sooner and creates more false positives; the right trade-off is workload-specific and should be configurable per node, not only globally.

L5. Fail-open needs a live target. With both backends dead, fail-open accepted the connection, kept an unhealthy node active, and closed after 1,019.21 ms with zero bytes. A client would have been better served by an immediate refusal. Fail-open is only meaningful if there is a defined fallback path (a

Table 11: Forwarding cost of the proxy path against a direct connection to the same backend on loopback. Persistent = one connection reused for $n = 500$ short requests; new-connection = a fresh connection per request ($n = 100$); bulk = 32 MiB echoed through one connection; burst = 100 simultaneous connections.

Workload	direct	gateway	ratio	note
TCP persistent RTT p50 (μs)	38.3	67.0	1.749×	one extra hop
TCP new-conn RTT p50 (ms)	1.34	10.18	7.624×	accept + connect pair
UDP request RTT p50 (μs)	36.1	66.1	1.831×	session lookup
Bulk 32 MiB throughput (MiB/s)	802.9 MiB/s	24.9 MiB/s	0.031×	8.4% CPU

Additional	Value
100 simultaneous connections served	100 succeeded, 75.4 ms wall (p50 32.6 ms)
resident set size, end of run	7.57 MB
idle probe traffic	9.08 probes/node/s

static page, a cache, a different pool); otherwise it converts a fast failure into a slow one.

L6. The admin plane is part of the attack surface. The parser handled garbage and injection-shaped input correctly, but it has no authentication and no line-length limit: a 65,536-byte line produced a 65,645-byte response. The loopback default contains the risk. Any deployment that binds the admin port elsewhere needs authentication, a line cap, and likely a separate network namespace. This is not a flaw in the failover logic; it is a reminder that a local operator interface becomes a remote vulnerability the moment its bind address changes.

6 Related Work

L4 proxies and load balancers. HAProxy (HAProxy Technologies), Envoy (Envoy Project), and NGINX (NGINX) all provide active and passive health checking, connection draining, and a choice of balancing policies, with far more configuration surface and L7 features than Sniffgate. Maglev (Eisenbud et al., 2016) shows how a software load balancer can be made fast and reliable at scale, using consistent hashing and a distributed control plane. Sniffgate is deliberately smaller: one process, static node lists, no TLS, no L7, and no distributed state. Its contribution here is not a new balancing algorithm but a measurement of what a compact implementation actually does at failure boundaries, including the parts its documentation gets wrong.

Failover protocols. VRRP (Nadas, 2010) and BFD (Katz and Ward, 2010a,b) solve failure detection at the network layer, with sub-second timers and strong operational experience. They move addresses or report path liveness; they do not own application sockets, drain TCP connections, or track per-protocol active sets. Sniffgate operates one level up, where it can see a client connection and a UDP session, at the cost of being a user-space process rather than a network primitive.

Failure detection. The failure-detector literature makes the trade-off between completeness and accuracy explicit (Chandra and Toueg, 1996), and accrual detectors replace binary timeouts with a continuous suspicion level (Hayashibara et al., 2004). Sniffgate uses the simplest binary detector: consecutive failures and successes with fixed thresholds. Our measurements show the cost of that simplicity: detection time is quantized by the probe timeout and the threshold, and a gray failure is indistinguishable from a crash. A production system might adopt an accrual detector or an adaptive timeout; a compact one can at least make the timeout per node.

Gray failure and partial failure. Huang et al. named gray failure as the gap between what a system observes and what its users experience (Huang et al., 2017). Our gray experiment is a minimal instance: the listener is up, the probe connects, and the response never comes. The gateway's only signal is the probe timeout, and the client's only signal is the request timeout; the two are not the same. The broader lesson—that differential observability, not just liveness, is needed—matches the gray-failure argument, though Sniffgate does not implement differential probing.

Replication, tail latency, and end-to-end arguments. State-machine replication (Schneider, 1990) and tail-tolerant techniques (Dean and Barroso, 2013) operate at layers above a failover gateway; Sniffgate assumes backends are stateless or replicate themselves. SRE practice (Beyer et al., 2016) emphasizes explicit error budgets and graceful degradation, which is exactly what the fail-open measurement lacks. The end-to-end argument (Saltzer et al., 1984) applies to health checks too: a probe that does not exercise the end-to-end path (here, the UDP path) cannot certify it.

Rust async runtime. The implementation builds on Tokio (Tokio Project) for scheduling and I/O and on `arc-swap` (arc-swap Contributors) for snapshot publication. The measured overhead is consistent with one userspace copy in each direction and a scheduler wake-up per direction; the design does not attempt kernel bypass, io_uring, or a custom allocator.

802.9

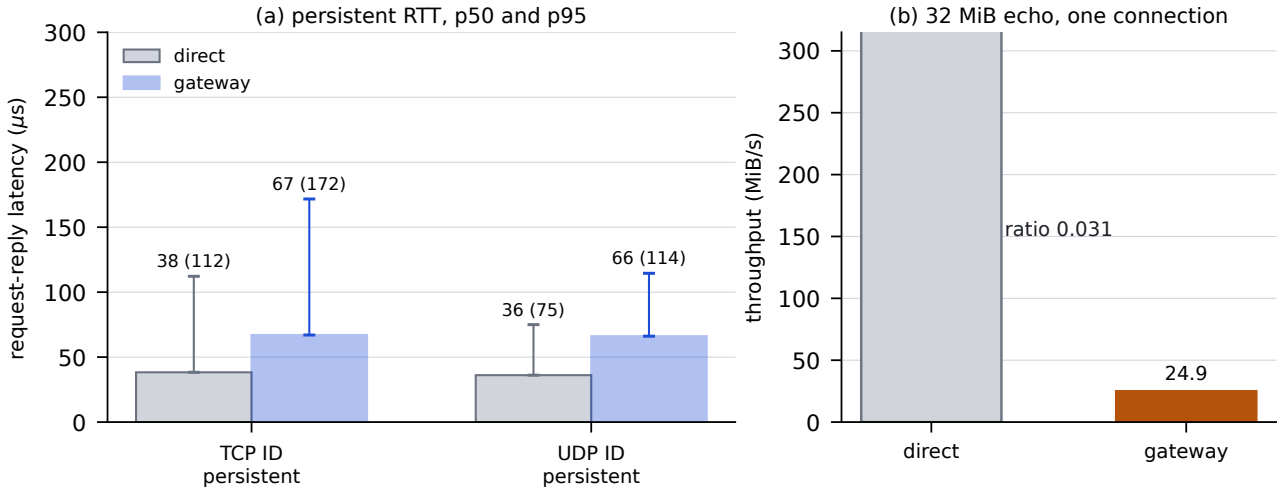


Figure 4: Forwarding overhead. (a) Persistent request–reply RTT on loopback, p50 with p95 whiskers. (b) One 32 MiB echo stream; the gateway reaches 0.031 of direct throughput because one core saturates on copying.

7 Limitations

The measurements are from one Windows 11 host over loopback, with echo backends and a Python client. Windows drops SYNs to closed TCP ports and resets closed UDP sockets immediately; a Linux host would likely produce a much faster TCP-probe arm and a different gray-failure ratio. The detection numbers should be read as a demonstration of the mechanism, not as constants for other hosts.

Sample sizes are small: one observation for startup and each reload step, two to five trials for failover, six client attempts for passive ejection. The first-request latency is one sample per failover trial. There are no confidence intervals, no controlled CPU load, and no repetition across reboots. The 1 ms timer bounds the resolution of short intervals, and the 4 s CPU windows make idle-CPU readings coarse.

The workload is minimal. There is no TLS, HTTP parsing, WAN latency, packet loss, reordering, NAT, or multi-host routing. The UDP switch experiment is a loopback burst, not a lossy network. The bulk test measures one core and one copy path, not a forwarding capacity. The 100-connection concurrency test is a smoke test, not a benchmark.

The gateway itself is a single point of failure. It has no peer, no state replication, and no self-failover; if the process dies, the service is down. The admin plane has no authentication and no line-length cap; the default loopback bind is the only control, and the hostile-input test is four cases, not fuzzing. The paper evaluates the implementation as it stands, including the places where it contradicts its README; it does not claim that those behaviors are intentional.

8 Conclusion

Sniffgate is a compact TCP/UDP failover gateway whose design fits in a few ideas: immutable snapshots, per-node probes,

passive ejection, deadline-based draining, and a line-oriented admin plane. Measuring it against real failures produced both the expected and the unexpected. The expected: a switch takes about a second with TCP probes on Windows, recovery takes about 195.24 ms, reload converges in about a quarter second, and forwarding costs roughly 1.749× latency and 0.031× bulk throughput. The unexpected, and more useful: detection is a property of the probe transport and the failure manifestation; `probe = none` silently disables passive ejection and can resurrect a dead node; draining is scoped to the union of protocol roles, so a TCP-only switch never drains; a UDP-only failure is invisible to a TCP probe; and fail-open can turn a refusal into a one-second hang. Each of these is a small implementation decision with an outsized operational consequence, and each was found by measuring the artifact rather than reading the feature list.

Data Availability

The gateway source is at commit `1793344` (2026-10-04). The measurement artifact, harness, figure script, and table script are in `papers/sniffgate/scripts/`. The artifact records the host, the Rust compiler, the binary SHA-256, every raw trial, and every aggregate used here. To reproduce the measurements on Windows with the same toolchain:

```
python scripts/failover_measure.py \
  --binary target/release/sniffgate.exe \
  --repo ../sniffgate
python scripts/make_figures.py
python scripts/make_tables.py
```

The first command rewrites the measurement artifact; the other two regenerate the figures and the LaTeX numbers and tables. A run takes several minutes because each scenario starts fresh processes and waits for probe transitions.

References

arc-swap Contributors. arc-swap: Atomically swappable Arc. https://docs.rs/arc-swap/. Accessed 2026-10-21.

B. Beyer, C. Jones, J. Petoff, and N. R. Murphy. *Site Reliability Engineering: How Google Runs Production Systems*. O'Reilly Media, 2016.

T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, 1996. doi: 10.1145/226643.226647.

J. Dean and L. A. Barroso. The tail at scale. *Communications of the ACM*, 56(2):74–80, 2013. doi: 10.1145/2408776.2408794.

W. Eddy. Transmission control protocol (TCP). RFC 9293, Aug. 2022.

L. Eggert, G. Fairhurst, and G. Shepherd. UDP usage guidelines. RFC 8085, Mar. 2017.

D. E. Eisenbud, C. Yi, C. Contavalli, et al. Maglev: A fast and reliable software network load balancer. In *Proceedings of the 13th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. USENIX Association, 2016.

Envoy Project. Envoy Proxy documentation. https://www.envoyproxy.io/docs. Accessed 2026-10-21.

HAProxy Technologies. HAProxy configuration manual. https://docs.haproxy.org/. Accessed 2026-10-21.

N. Hayashibara, X. Défago, R. Yared, and T. Katayama. The phi accrual failure detector. In *Proceedings of the 23rd IEEE International Symposium on Reliable Distributed Systems (SRDS)*. IEEE, 2004.

P. Huang, C. Guo, L. Zhou, et al. Gray failure: The Achilles' heel of cloud-scale systems. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems (HotOS)*. ACM, 2017.

D. Katz and D. Ward. Bidirectional forwarding detection (BFD). RFC 5880, June 2010a.

D. Katz and D. Ward. Bidirectional forwarding detection (BFD) for IPv4 and IPv6 (single hop). RFC 5881, June 2010b.

Keepalived Project. Keepalived. https://www.keepalived.org/. Accessed 2026-10-21.

Microsoft. Windows sockets error codes (Winsock2.h). https://learn.microsoft.com/en-us/windows/win32/winsock/windows-sockets-error-codes-2. Accessed 2026-10-21.

S. Nadas. Virtual router redundancy protocol (VRRP) version 3 for IPv4 and IPv6. RFC 5798, Mar. 2010.

NGINX. NGINX documentation: Http health checks. https://docs.nginx.com/nginx/admin-guide/load-balancer/http-health-check/. Accessed 2026-10-21.

J. Postel. User datagram protocol. RFC 768, Aug. 1980.

J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems*, 2(4):277–288, 1984. doi: 10.1145/357401.357402.

F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, 1990. doi: 10.1145/98163.98167.

Tokio Project. Tokio: An asynchronous Rust runtime. https://tokio.rs/. Accessed 2026-10-21.