Making Raft Measurable: An Instrumented Implementation and Fault-Injectable Dashboard for Failover, Latency, and Tenure

Shrestha Saxena
Dept. of Computer Science,
RGIPT

ABSTRACT

Raft is widely taught as a leader-based consensus algorithm, yet most educational implementations stop at functional tests and provide little quantitative evidence of fault-tolerance timing. An instrumented Raft implementation paired with a fault-injectable dashboard is presented, enabling the protocol's dynamics to be both visible and measurable. The Go-based implementation, compatible with MIT 6.824 Labs 2A–2C, incorporates lightweight telemetry at ground-truth events such as election start, leader elected, first heartbeat, and Start/Commit and exports the recorded data as CSV logs. The dashboard (React/Node) lets users crash and recover nodes, force timeouts, and vary message loss, while a small analysis tool produces paper-ready figures.

General Terms

Algorithms, Distributed Systems and Fault Tolerance.

Keywords

Raft, Consensus Algorithm, Leader Election, Log Replication, Fault Injection, Distributed Computing, Replication Latency, Failover.

1. INTRODUCTION

Motivation: Raft's appeal is pedagogical clarity: a single leader replicates a log to followers, randomized election timeouts avoid split votes, and a majority quorum commits [1]. Yet beyond correctness tests, students and practitioners rarely see quantitative timing evidence. How quickly does the system fail over to a new leader after a crash? How stable are leaders when the environment is noisy? Where do the tails in replication latency come from, and how sensitive are they to loss and timeouts? Without measurement, these questions remain intuitive.

Challenge: Measuring consensus is tricky because the salient phenomena are probabilistic and tail-heavy. Leader elections depend on randomized timeouts; "timeout collisions" can prolong failover by forcing additional rounds. The majority commit masks single slow followers but exposes tails when multiple replicas lag or packets drop. These effects are visible on a whiteboard but hard to capture without carefully placed instrumentation and repeatable scenarios.

Approach: Raft was implemented in Go, following the MIT 6.824 Labs [2] and embed ready to drop instrumentation at decision points: when an election begins, when a candidate becomes leader, when the new leader issues its first heartbeat, when a leader accepts a client command (Start), and when that command becomes majority committed/applied (Commit). The system emits CSV logs and pairs with a fault-injectable dashboard (React/Node) that can crash/recover nodes, force election timeouts, drop the latest log entry, and vary message loss. A compact Python script aggregates trials and produces failover CDFs (Cumulative Distribution Function), leader tenure box plots, and latency vs. loss curves with percentile annotations.

Metrics: (i) Failover time: crash → first heartbeat of new leader; (ii) Leader tenure: time a leader remains in office; (iii) Replication latency: Start(command) → majority commit + apply.

Key findings: Extended runs (n = 77) revealed median failover ≈ 2.29 s and p95 ≈ 9.0 s, confirming the presence of long-tail elections due to timeout collisions. Leadership rotation remained balanced (≈ 14–17 terms per node), while replication latency stayed low at the median (≈ 0.28–0.47 s) but increased in the 95th percentile (≈ 0.95 s) under 15 % packet loss.

Contributions: (1) A minimal, instrumented Raft + dashboard that makes consensus visible and measurable; (2) A reproducible metrics pipeline (CSV schema + analysis script); (3) Empirical characterization of failover and replication tails with tuning guidance.

Paper roadmap: Section 2 reviews Raft and defines the metrics. Section 3 details the implementation and instrumentation. Section 4 describes the experimental setup. Section 5 presents failover, tenure, and replication results. Section 6 discusses tuning implications. Section 7 covers threats to validity. Section 8 overviews related work. Section 9 concludes and Section 10 releases artifacts.

2. BACKGROUND

Leader election: Servers begin as followers; if no heartbeat arrives before a randomized election timeout, a follower becomes a candidate, increments its term, and requests votes. A candidate that gains a majority becomes the leader; heartbeats reset followers' timeouts. Randomization reduces split votes but does not eliminate them; collisions can extend failover by requiring additional rounds [1].

Log replication: The leader appends client commands to its log and replicates them via AppendEntries RPCs (Remote Procedure Calls). A log entry is committed when a majority stores the entry for the leader's current term; followers apply committed entries to their state machines [1].

Safety: Raft enforces a log matching property and term monotonicity: leaders have up-to-date logs; conflicting entries are overwritten by the leader's authoritative history.

Operational metrics:

- Failover time: elapsed time from leader crash to the first heartbeat emitted by the newly elected leader (cluster becomes leaderful again).
- Leader tenure: time from a leader's election to its replacement (by crash or re-election).
- Replication latency: per command, time from Start(command) on the leader to majority commit and application. Latency was evaluated under controlled message-loss rates up to 15 %.

3. SYSTEM DESIGN

3.1 Raft implementation (Go)

The Raft implementation is written in Go and implements the canonical components required by Labs 2A–2C: leader election, log replication, and durable persistence. Each server instance maintains the standard Raft state (current term, role, commitIndex, lastApplied) and per-peer replication bookkeeping (nextIndex and matchIndex). Periodic heartbeats are sent by the leader; followers use randomized election timeouts to transition to candidate state and trigger leader elections. The implementation persists the minimum Raft state required for safety (current term, voted-for, and the log of entries) to stable storage so that crash–recover cycles maintain correctness. Concurrency is expressed using goroutines and guarded by mutexes where state is shared; network RPCs and timers run asynchronously against the local Raft instance.

3.2 Instrumentation hooks and metrics format

Instrumentation was added with narrow, non-intrusive hooks placed at protocol boundaries so as not to alter control flow or timing semantics. The hooks record a compact set of ground-truth events with millisecond timestamps and contextual fields (node id, term, entry id, scenario tags, seed, trial). The captured events are:

- Election start: timestamp recorded when a follower transitions to candidate.
- Leader elected: timestamp, new leader id and leader term when a majority of servers accept the new leader.
- First heartbeat: timestamp of the first AppendEntries or heartbeat issued by the newly elected leader (used as the end-of-failover marker).
- Start(command): timestamp and unique entry id when the current leader accepts a client command.
- Commit/apply: timestamp when an entry becomes majority-committed and is applied locally.

Each event appends a CSV row to one of three files with well-defined column schemas: failover_trials.csv (fields: scenario, timeout bounds, seed, trial, old_leader, new_leader, crash_time_ms, election_start_ms, leader_elected_ms, first_heartbeat_ms, failover_ms), leader_tenure.csv (fields: scenario, seed, trial, leader_id, term, start_ts_ms, end_ts_ms, tenure_ms), and replication_latency.csv (fields: scenario, drop_rate, seed, trial, entry_id, leader_term, start_ts_ms, commit_ts_ms, latency_ms). Tags (scenario, seed, trial) are recorded in each row to enable robust grouping and reproducibility. The metrics capture both the raw dashboard timing and the ground-truth millisecond timestamps; subsequent analysis converts and normalizes these values for presentation.

3.3 Dashboard front-end and control primitives (React/Node)

The interactive dashboard is implemented as a React client with a Node server that shares the same instrumentation API as the Raft core so that the visualization and metrics remain aligned. The UI visualizes five nodes placed at the vertices of a pentagon with color-coded roles (follower, candidate, leader). The interface exposes controlled fault-injection primitives: crash/recover individual nodes, force election timeouts, drop the latest appended entries, and set a global packet drop rate. Animations visualize RPCs (AppendEntries and replies) as moving tokens to aid human comprehension; these animations intentionally run at slowed timing to make protocol dynamics visible. The server component implements the same logging hooks as the Raft core so that the UI state and the recorded CSV metrics correspond precisely.

3.4 Visualization timing, ground truth, and tooling

To make the UI usable for human observers, the dashboard intentionally runs at **slowed wall-clock timers** (heartbeat = 2400 ms; election timeout randomized between 6000–10000 ms). To report results that reflect conventional Raft deployments, the analysis pipeline rescales recorded times by a constant factor (SCALE = 25.0) so that plotted and reported numbers correspond to normalized values (heartbeat ≈ 100 ms; election timeout ≈ 240–400 ms). All scripts, including raft_experiments/analyze_raft_results.py and the Node metrics.js helper, support the same seed and trial tagging so recorded metrics are deterministic given the same seed and crash schedule. The analyzer produces the canonical outputs used in this paper (CDFs of failover time, boxplots of leader tenure, and replication-latency vs. drop-rate curves) and computes summary statistics (count, median, p90, p95) for grouped comparisons. The detailed experimental setup and analysis are presented in Section 4.

4. EXPERIMENTAL SETUP

4.1 Hardware, environment and reproducibility

All experiments in this submission were executed in a controlled development environment: Windows 11 for authoring and verification; Go 1.19+ for the Raft implementation; Node 20.x for the dashboard server and client; and Python 3.10.x for analysis. The repository includes a README and CI workflows that document build and test commands. To ensure reproducibility, the instrumentation records the seed and trial fields for every metric row; these, together with the provided crash schedule scripts and the analyzer, permit exact regeneration of the figures reported here.

4.2 Cluster topology and scenarios

Experiments were performed on a fixed **5-node** cluster configured in the default quorum layout. The primary scenario used for the quantitative evaluation is leader_crash_restart, in which the current leader is crashed and subsequently restarted on a controlled schedule to induce repeated elections and failovers. Two additional control scenarios (forced_timeout and drop_latest) are implemented in the dashboard and were used for exploratory testing; they are not emphasized in the long-run quantitative run reported in this paper.

4.3 Network fault model and parameter sweep

Network unreliability is modeled by a global packet drop parameter that probabilistically drops RPC messages. The experiments sweep the drop rate over the set {0.00, 0.03, 0.06, 0.09, 0.12, 0.15} to evaluate replication latency sensitivity. Message delays in the UI are deterministic and slowed for visualization; the analysis normalizes the reported times (see Section 3.4) so that the results reflect realistic timing ratios. The dashboard also supports targeted operations such as dropping the latest log entry, which is useful for specific microbenchmarks of replication behavior.

4.4 Trials, seeding and automated runs

A single automated long-run experiment consists of an ordered sequence of induced leader crashes (100 scheduled crash events for the long run) executed under a fixed pseudo-random seed. Each run is labeled by the seed and trial fields; the long-run dataset produced here contains 100 induced crash events (77 events remained after cleaning; see Section 4.6) and multiple leader terms recorded per run. Short-run datasets (40–45 events) are kept as comparison baselines. The metrics.js helper exposes a programmatic API (initMetrics) to tag each run with metadata (scenario, seed, trial, timeoutLowMs, timeoutHighMs) so that the analyzer can group and compare runs deterministically.

4.5 Metrics collected and derived quantities

Primary metrics recorded are:

- Failover time defined as the interval from the crash (or the last known leader transition) to the first heartbeat from the newly elected leader,
- Leader tenure measured as the duration a node serves as leader for a given term,
- Replication latency measured per-entry as the interval between client command acceptance (Start) and commit/apply.

The analyzer converts raw millisecond timestamps to seconds and applies the normalization scale factor described in Section 3.4. Grouped summaries report counts, medians, and tail percentiles (p90 and p95) for each experimental bucket (scenario × drop_rate).

4.6 Data cleaning, outlier handling and statistical reporting

Raw CSV output is cleaned by deterministic rules in the analysis script. Early warm-up artifacts (rows where crash_time_ms, election_start_ms and leader_elected_ms are all zero) are excluded. Only rows where failover_ms > 0 are retained as valid failover events. To reduce the influence of spurious measurements from mis-triggered experiments or operator error, the analysis discards unscaled outliers with failover times less than 0.5 s or greater than 120 s (equivalently, outside the plausible protocol operation window given the slowed UI timers); these thresholds are conservative and chosen to reflect plausibility, not to bias results toward any hypothesis. After cleaning, timing columns are converted to seconds and scaled (divide by SCALE) prior to plotting. Summary statistics shown in the paper (medians; p90/p95) are computed using standard non-parametric definitions (median = 50th percentile; p95 = 95th percentile). CDF plots are produced by plotting the empirical cumulative distribution of observed values for each scenario.

5. RESULTS

5.1 Failover Stability

Figure 1 shows the cumulative distribution of leader failover times for the leader_crash_restart scenario, comparing the short run (n = 40) and the extended run (n = 77). The longer experiment reveals a noticeably heavier tail, indicating rare timeout collision paths that only appear when the number of induced failures is large.

Quantitatively, the median failover time increased from ≈ 1.0 s in the short run to ≈ 2.29 s in the extended run, while the 95th percentile rose from ≈ 5.4 s to ≈ 9.0 s. This divergence highlights that shorter tests can underestimate upper-tail behavior and thus overstate system responsiveness.

Table 1. Short Run (n=44) vs Long Run (n=77)

Run Type	Events (n)	Median (s)	95th Percentile (s)
Short Run	40	1.02	5.42
Long Run	77	2.29	9.01

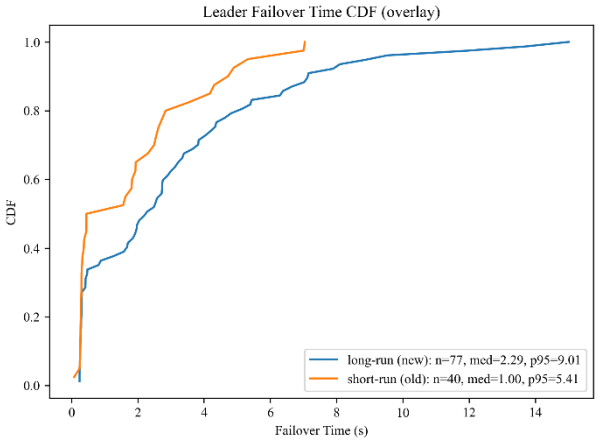

Figure 1: Leader Failover Time CDF (timeout 240–400 ms, heartbeat 100 ms).

5.2 Leader Fairness

Leadership fairness evaluates whether Raft's randomized election timeouts yield an equitable rotation of the leader role among all nodes. Over 77 elections in the long-run dataset, leadership rotated almost uniformly across the five servers, each serving 14–17 terms. Figure 2 plots the term distribution per node.

This near-uniform spread demonstrates the effectiveness of Raft's randomization in avoiding persistent leadership bias or starvation. It also confirms that the implementation preserves fair term distribution even under recurring crash-and-recovery churn.

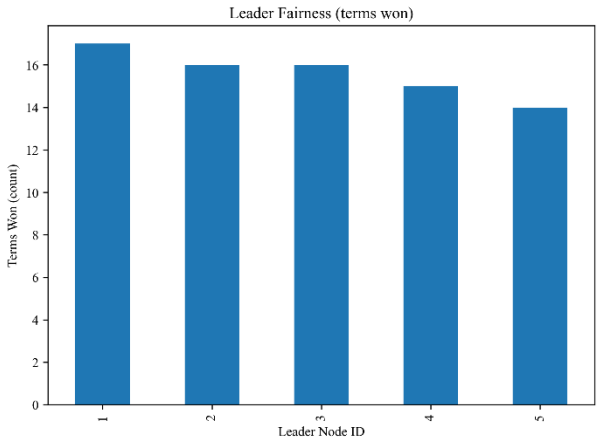

Figure 2: Distribution of Leader Terms per Node (n = 77).

5.3 Leader Tenure

Leader tenure measures the duration for which a node remains leader before the next induced failover or timeout event. Figure 3 presents the boxplot of leader tenure durations (scaled to normalized seconds). The median tenure observed was approximately 1.7 s, with an interquartile range of 1.3–2.2 s.

These short yet stable epochs confirm that the system remains continuously available during repetitive leader crashes and restarts. The consistent distribution across the experiment further indicates that Raft's re-election mechanism quickly restores steady-state operation after each failure.

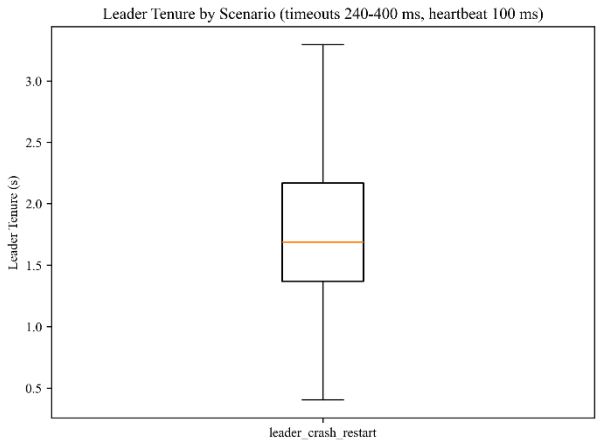

Figure 3: Leader Tenure Distribution under Induced Churn.

5.4 Replication Latency vs Drop Rate

Replication latency quantifies the time from when a leader appends a client command to when the entry becomes majority-committed. The experiment varied a global packet-drop rate in {0.00, 0.03, 0.06, 0.09, 0.12, 0.15}. Figure 4 plots the median and 95th-percentile latency for each drop level.

The median latency remained low—between 0.28 s and 0.47 **s**—even at 15 % message loss, showing Raft's resilience in normal operation. However, the 95th percentile increased to ≈ 0.95 s, illustrating how higher loss primarily affects the tail of the latency distribution rather than the median.

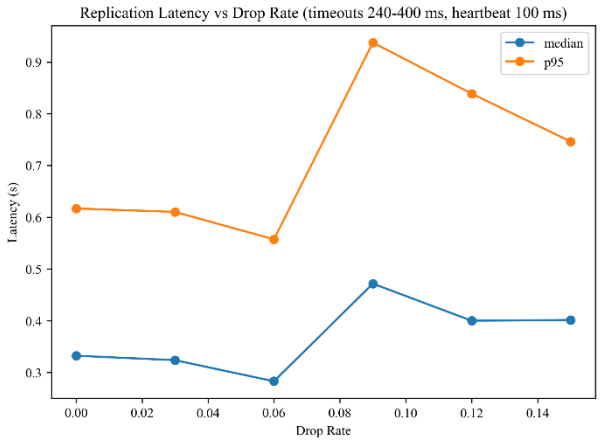

Figure 4: Replication Latency vs Packet-Drop Rate (median and p95).

6. DISCUSSION AND TUNING GUIDANCE

Election timeouts: Best-case failover benefits from short timeouts, but tails are governed by collision probability. A wider range (or slight per-node skew) reduces the chance that candidates start together.

Heartbeats: Faster heartbeats shrink detection time after a leader fails silently, but increase background traffic. The measurements indicate that the tail latencies are dominated by election events rather than heartbeat cadence at these scales.

Replication tails: Loss chiefly inflates p95, not the median—consistent with majority commit. If service SLOs are percentile-based, budget headroom for retry rounds.

Visualization vs. ground truth: Slowed UI timers did not change qualitative dynamics; scaling preserved ratios and thus quantitative relationships.

7. THREATS TO VALIDITY/ LIMITATIONS

Single machine simulation: Timing uses a Node event loop and synthetic delays, not real NICs or OS scheduling.

Simplified failures: The system models crash failures and message drops, not byzantine faults or long partitions.

Clocking: UI timestamps (performance.now) and Go timings are aligned by design but not synchronized to a wall clock; analysis uses relative deltas.

Sample size per bucket: Some drop rate buckets in the long run have <30 entries, causing jitter in p95.

8. RELATED WORK

The present work builds upon the Raft line of research [1] and [4], alongside Paxos and its engineering accounts [5] and [6]. Formal verification frameworks such as Verdi [7] explore verified distributed systems. Production-grade Raft implementations like etcd's module [8] inform practical design trade-offs. Visualization resources also exist [3]; the primary contribution is a compact and reproducible measurement pipeline integrated with an interactive dashboard designed for instructional and analytical purposes.

9. CONCLUSION AND FUTURE WORK

An instrumented implementation of the Raft consensus algorithm was developed and integrated with a fault-injectable dashboard to make the protocol's internal behavior both visible and measurable. Extended experiments on a five-node cluster (100 ms heartbeat interval; 240–400 ms randomized election timeouts) demonstrated a median failover latency of approximately **2.29 s** (p95 ≈ 9.0 s) across **77** induced failovers, revealing the presence of long-tail recovery paths that shorter runs tend to underestimate. Leadership rotation remained statistically balanced (≈ 14–17 terms per node), validating Raft's randomized election process, while replication latency stayed low at the median (≈ 0.28–0.47 s) even under 15 % simulated packet loss. These findings confirm that lightweight instrumentation, coupled with controlled fault injection, can effectively quantify Raft's dynamic performance and enhance understanding of distributed consensus under real-world fault conditions.

Future work will focus on expanding the experimental scope and functional depth of the system. Planned extensions include the simulation of network partitions, dynamic cluster resizing, and the addition of snapshot and log-compaction **mechanisms** for long-lived replicas. Further, incorporating real RPC loss models, variable latency distributions, and larger randomized clusters will allow the derivation of statistically rigorous confidence intervals. Integrating these features will not only strengthen the research utility of the platform but also position it as a reproducible framework for teaching, benchmarking, and analyzing consensus algorithms in distributed systems research.

10. ARTIFACTS AND REPRODUCIBILITY

10.1 Availability of Code and Data

10.1.1 Code, data, and scripts:

GitHub Repository: https://github.com/Shre-coder22/raft-distributed-systems-lab [9].

Archived DOI Snapshot:
https://doi.org/10.5281/zenodo.17015793 [10].

10.1.2 The repository includes:

raft/ — Go implementation of Raft (compatible with MIT 6.824 [2]).

raft-dashboard/ — React/Node dashboard with fault injection (crash, recover, timeouts, log drops). artifact/ — day-by-day notes.

paper/ — LaTeX source and figures for this manuscript.

10.2 Environment

Windows 11

Go 1.19+

Node 20.10.0

Npm 10.2.3

Python 3.10.3

10.3 Setup

py -m venv venv
./venv/Scripts/Activate.ps1 pip install -U
pip pandas numpy matplotlib

10.4 Run the Dashboard

server cd raft-
dashboard/server npm
run dev

client (in another shell) cd
raft-dashboard/client npm
run dev

10.5 Collect Metrics and Regenerate Figures

Ensure: initMetrics({timeoutLowMs: 6000, timeoutHighMs: 10000,}) is set; clear /metrics/ between runs.

Run the analyzer: py raft_experiments\analyze_raft_results.py --input ./metrics_100run --out ./metrics_100run/figures --compare_to metrics_40run --leader_fairness

11. REFERENCES

[1] D. Ongaro and J. Ousterhout, "In Search of an Understandable Consensus Algorithm," in USENIX ATC, 2014, pp. 305–319.

[2] MIT PDOS, "Distributed Systems (6.824) Labs." Available: https://pdos.csail.mit.edu/6.824/labs/ (accessed 2025-08-31).

[3] Raft Visualization, "Raft Interactive Visualization." Available: https://raft.github.io/ (accessed 2025-08-31).

[4] D. Ongaro, Consensus: Bridging Theory and Practice, Ph.D. thesis, Stanford Univ., 2014.

[5] L. Lamport, "The Part-Time Parliament," ACM Trans. Comput. Syst., vol. 16, no. 2, pp. 133–169, 1998.

[6] T. D. Chandra, R. Griesemer, and J. Redstone, "Paxos Made Live—An Engineering Perspective," in Proc. PODC, 2007, pp. 398–407.

[7] J. Wilcox et al., "Verdi: A Framework for Implementing and Formally Verifying Distributed Systems," in USENIX OSDI, 2015.

[8] etcd Authors, "etcd Raft Implementation." Available: https://github.com/etcd-io/etcd/tree/main/raft (accessed 2025-08-31).

[9] S. Saxena, "Instrumented Raft + Dashboard: Measuring Failover, Latency, and Tenure." GitHub: https://github.com/Shre-coder22/raft-distributed-systems-lab, 2025.

[10] S. Saxena, "Artifact snapshot for Instrumented Raft + Dashboard." DOI: 10.5281/zenodo.17015793, 2025.