The Journal of Open Source Software

# Making Raft Visible: an instrumented consensus implementation with a fault-injectable dashboard

**Shrestha Saxena** [1]

**1** Department of Computer Science and Engineering, Rajiv Gandhi Institute of Petroleum Technology (RGIPT), Jais, Uttar Pradesh, India

**DOI:** 10.xxxxxx/draft

**Software**

- Review
- Repository
- Archive

**Editor:** Daniel S. Katz

**Reviewers:**

- @srbwin1
- @abhishektiwari

**Submitted:** 09 September 2025
**Published:** unpublished

## Summary

Raft is a leader-based consensus protocol for state machine replication: a cluster elects a single leader via randomized timeouts; clients send operations to the leader, which appends them to a replicated log and commits entries once a majority acknowledges. Safety is ensured by leader election and log-matching rules; liveness is maintained under crash faults and unreliable networks through heartbeats and timeouts. Raft targets the same fault model and efficiency as Multi-Paxos while being simpler to understand and implement (Lamport, 1998; Ongaro & Ousterhout, 2014).

Raft is widely taught, yet its timing behavior under failures is rarely quantified outside production systems. This package provides a compact, instrumented Raft implementation plus a browser dashboard for **fault injection** (leader crashes, packet loss) and **ground-truth logging** (election start, leader elected, first heartbeat, replication commit). The toolkit turns classroom Raft into a **measurable system**: users run scripted experiments and regenerate all plots from CSV metrics.

**Illustrative use of the software.** The package exposes a fault-injection workflow that orchestrates leader crashes and packet-loss regimes, timestamps ground-truth events (election start, leader elected, first heartbeat, replication commit), and regenerates figures from the resulting CSV logs. The repository includes scripted runs and small example datasets that exercise this pipeline end-to-end. The numbers shown in the example figures are the direct output of the software's default scenarios; their role is to demonstrate functionality and reproducibility of the artifact. Users can vary seeds, timeouts, and drop rates (or add new scenarios) to obtain their own measurements.

## Statement of need

Researchers and instructors have visual demos of Raft, but lack **open, reproducible** tools to **measure** failover and replication under controlled churn. This package fills that gap with (i) an instrumented Raft core, (ii) a UI that injects faults and records events, and (iii) an analysis pipeline that yields CDFs, tenure distributions, and latency-vs-loss plots. It targets courses, benchmarking, and practitioners exploring timeout/SLO trade-offs in consensus systems (Lamport, 1998; Ongaro & Ousterhout, 2014).

## State of the field

Production Raft systems (e.g., etcd) and formal verification frameworks (e.g., Verdi) emphasize **correctness** and safety proofs (*Etcd*, n.d.; Wilcox et al., 2015). Educational visualizations (e.g., Stanford's Raft viz; MIT 6.824 labs) illustrate protocol dynamics but generally lack

**ground-truth metrics** and **reproducible long-run** experiments (*MIT 6.824/6.5840 Distributed Systems Labs*, n.d.; *Raft Visualization*, n.d.). Our contribution is a lightweight **measurement pipeline** complementary to these efforts.

## Functionality and quality control

The repository provides three components:

- **Instrumented Raft core** (`raft/`) with event hooks for elections, heartbeats, and replication.
- **Fault-injectable dashboard** (`raft-dashboard/server/`, `raft-dashboard/client/`) to trigger leader crashes and packet-drop rates and to visualize state.
- **Metrics and analysis** (`raft-dashboard/server/raft_experiments/`) that log ground-truth events to CSV and regenerate figures (CDFs, leader-tenure, latency-vs-loss).

Quality control:

- **Tests:** Go unit tests for election and log-matching invariants (e.g., `go test -run 2A`), plus `go test ./...`.
- **CI:** GitHub Actions workflow runs tests/lint on push and PRs.
- **Reproducibility:** Timers in the UI are slowed for pedagogy; the analyzer normalizes times by 25x to reflect common practice (heartbeat ~100 ms; election timeout 240–400 ms), preserving ratios and Raft dynamics. Fixed seeds and deterministic crash schedules regenerate the example figures.

## Installation & quick start

This package requires three dependencies: **Go (version 1.19 or newer)**, **Node.js (version 20 or newer)**, and **Python (version 3.10 or newer)**. Once these are installed, the software can be obtained directly from GitHub by cloning the repository and moving into the project directory:

```
git clone https://github.com/Shre-coder22/raft-distributed-systems
cd raft-distributed-systems
```

The Raft dashboard has two components, a backend server and a frontend client, which must be started separately. To launch the server, open a terminal, install the dependencies, and start the development process:

```
cd raft-dashboard/server
npm install
npm run dev
```

In a separate terminal, start the client in the same way:

```
cd raft-dashboard/client
npm install
npm run dev
```

Once both server and client are running, the dashboard will open in the browser. This interactive interface allows control over Raft nodes, leader elections, and injected faults.

For analysis, the repository includes an example dataset from a 100-step experiment. The analysis script processes the recorded metrics and regenerates the main figures (failover distribution, leader tenure, and replication latency). To run the analyzer, execute:

```
cd raft-dashboard/server
py raft_experiments/analyze_raft_results.py --input ./metrics_100run --out ./metrics_100
```

(If your system does not recognize py, replace it with python.) The generated figures are written to ./metrics_100run/figures, reproducing those shown in the manuscript.

## References

*Etcd: Distributed reliable key-value store.* (n.d.). https://github.com/etcd-io/etcd.

Lamport, L. (1998). The part-time parliament. *ACM Transactions on Computer Systems*, *16*(2), 133–169. https://doi.org/10.1145/279227.279229

*MIT 6.824/6.5840 distributed systems labs.* (n.d.). https://pdos.csail.mit.edu/6.824/.

Ongaro, D., & Ousterhout, J. (2014). In search of an understandable consensus algorithm (raft). *USENIX Annual Technical Conference (ATC)*. https://www.usenix.org/conference/atc14/technical-sessions/presentation/ongaro

*Raft visualization.* (n.d.). https://thesecretlivesofdata.com/raft/.

Wilcox, J. R., Woos, D., Panchekha, P., Wang, Z., Tatlock, Z., Ernst, M. D., & Anderson, T. (2015). Verdi: A framework for implementing and formally verifying distributed systems. *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. https://doi.org/10.1145/2737924.2737958