A Cluster That Could Not Keep a Leader

A post-mortem on nexus_cluster: what happens when your runtime pauses for longer than your failure detector is willing to wait.

Abir Deol · University of Alberta, Computing Science · July 2026

Abstract. nexus_cluster is a Raft consensus implementation written from scratch in Python, with leader election, log replication and term management over a custom TCP transport, run as local processes and driven by a CLI. This is a post-mortem on a cluster that could not hold a leader. There was no network, no partition and no packet loss, because every node was on one machine. The election timeouts I had taken from the Raft paper were shorter than the pauses my own runtime was capable of producing, so the cluster spent its time deposing leaders that were merely busy.

1. How Raft decides a leader is gone

Raft has one leader per term. The leader sends periodic AppendEntries messages, empty ones acting as heartbeats. Each follower runs an election timeout, randomized within a range. If a follower hears nothing from the leader before that timeout expires, it assumes the leader has failed, increments its term, becomes a candidate and solicits votes.

The safety properties do not depend on the timeout value. Raft will not elect two leaders in a term or commit conflicting entries regardless of how the timers are set. What the timeout controls is liveness, and the paper is explicit about the requirement: the broadcast time must be well below the election timeout, which must in turn be well below mean time between failures. The paper suggests figures in the range of 150 to 300 milliseconds, and those numbers are grounded in the assumption that the dominant delay between a leader sending and a follower receiving is the network.

I implemented that faithfully, including the randomization, and copied the range. On a single machine with no network, that assumption does not hold, and I did not think about it until the cluster forced me to.

2. The symptom

Leadership churned. Bring up five nodes, watch a leader get elected correctly, and then watch it get replaced seconds later with no node having failed. Term numbers climbed steadily. Replication throughput was poor because the cluster was constantly in the middle of an election rather than doing work.

```
[node-2] elected leader        term 7
[node-4] election timeout      term 8   candidate
[node-4] elected leader        term 8
[node-1] election timeout      term 9   candidate
[node-3] election timeout      term 9   candidate
[node-1] elected leader        term 9
[node-2] election timeout      term 10  candidate
```

The behaviour was correct in the narrow sense. Every one of those nodes followed the protocol exactly: it did not hear a heartbeat in time, so it stood for election. Nothing in the implementation was doing anything other than what Raft says to do. The cluster was correctly responding to a false signal.

3. The hunt

My first assumption was a bug in the transport, because the alternative was that heartbeats were being sent and not arriving in time, and on localhost that seemed absurd. I checked for dropped connections and for heartbeats being generated at the wrong interval. Both were fine.

The measurement that settled it was timestamping the actual gap between consecutive received heartbeats at a follower, rather than the interval at which the leader intended to send them. Those are different numbers and the difference was the whole story. The intended interval was steady. The observed gaps had a long tail: mostly tight, and then occasionally several hundred milliseconds, which is past the election timeout.

Correlating those spikes with what the leader was doing at the time made the cause obvious. The long gaps lined up with serializing larger log batches and with garbage collection.

4. Root cause

The leader was not failing. It was pausing, and my timeouts could not tell the difference between those two things because the timeout was smaller than the pause.

Two sources, both of them properties of the runtime rather than of the algorithm. First, the interpreter executes bytecode under a global interpreter lock, so a thread doing a stretch of CPU-bound work in the same process does not interleave with the heartbeat path the way concurrent threads suggest it might. Serializing a batch of log entries is exactly that kind of work. Second, garbage collection pauses the process, and a Raft node under write load produces a great deal of short-lived garbage: message objects, serialized buffers, log entry copies.

The compounding factor is that every node lived on one machine and shared one CPU. A collection or a serialization burst on the leader is not an isolated event when the followers are contending for the same cores. The system had a strong tendency to be busy everywhere at once, which is the worst possible correlation for a failure detector.

So the honest statement of the bug is that I imported a constant along with an assumption and only noticed the constant. The 150 to 300 millisecond range encodes the belief that scheduling jitter is small relative to network latency. In a single process on one machine, there is no network latency, and the jitter is all that remains.

5. The fix

Measure first, then choose. The election timeout should sit above the observed tail of heartbeat delivery, not above the average.

- Raised the election timeout well clear of the measured worst case rather than the typical case, and kept the randomization so that simultaneous candidacies still break up.
- Held the heartbeat interval at a small fraction of the election timeout, so a single missed heartbeat is never on its own sufficient to trigger an election.
- Reduced the size of the work done inline on the heartbeat path, so that replicating a large batch does not delay the signal that says the leader is alive.
- Logged the delivery gap distribution rather than the intended interval, because the intended interval was never the thing that was wrong.

What I deliberately did not do is chase the timeout downward for its own sake. A shorter timeout buys faster recovery from a genuine leader failure and costs stability against false positives, and on this deployment the false positives were the entire problem. Choosing that number is a tradeoff about the environment, not a value to be copied.

6. What it changed

This is the one I think about most, because nothing was broken. The Raft implementation was correct. The transport was correct. The bug lived in a constant, and the constant was wrong because it belonged to a set of assumptions that did not survive the move from the paper to my machine.

- A failure detector cannot distinguish a crashed process from a slow one. That is a known result and I had read it, but reading it and watching your own cluster depose a healthy leader are different kinds of knowing.
- Timeout constants carry assumptions about the environment. Copying the number without copying the environment imports a belief you have not checked.
- Measure the tail, not the mean. The average heartbeat gap was fine throughout, and the average was irrelevant, because elections are triggered by the worst case.
- The runtime is part of the system. Interpreter pauses and collection pauses are latency exactly like network latency, and any protocol with a time-based failure detector has to be tuned against them.

The general form, which I now apply before assuming a distributed system is broken: ask what would have to be true for this behaviour to be correct. Here, every node was behaving impeccably given the information it had. The information was wrong, and it was wrong because of a threshold I chose without measuring the thing it was thresholding.