Raft Reconfiguration Operator — Team Plan (Bugs, Correctness, Evaluation)

Team: Mingjun Liu · Haosen Fang · Wei Luo
Course: CS 598 — Cloud Storage Systems
Target Runtime: kind/minikube on macOS (ARM64 OK)
Primary Language: Go (server + operator)

---

0) Executive Summary

We will build a **protocol-aware Kubernetes Operator** for a Raft-based KV store that makes reconfiguration **safe, automatic, and declarative**. The operator exposes a `RaftCluster` CRD and drives the *explicit* Joint Consensus sequence (AddLearner → CatchUp → EnterJoint → CommitNew → LeaveJoint) while enforcing invariants (learner non-voting, quorum intersection, safe leader transfer, readiness gating). The control loop will enforce the full set of Raft safety invariants: Learner non-voting until fully caught up (log prefix, snapshot, and term alignment) Quorum intersection preserved during EnterJoint and CommitNew. Leader safety via preStop hooks, leader transfer before removing the leader, and refusal to proceed if leadership is unstable. Readiness gating: promotion only allowed after caughtUp && durable. One reconfiguration in flight, with cooldowns based on .spec.reconfig.rateLimit. Monotonic configuration index: the operator ensures configuration entries never roll back or interleave incorrectly.

To motivate the design and to validate correctness, we will also implement **two unsafe baselines** that reproduce real safety violations during reconfiguration:

- **Bug 1: Early-Vote Learner** — a new member becomes a **voter** before catching up, enabling an out-of-date leader to win.
- **Bug 2: No Joint Consensus** — membership changes are applied as independent updates (remove+add) without a joint phase, enabling **non-intersecting quorums** and conflicting commits.

We will compare correctness (Porcupine linearizability) and availability (failover downtime, reconfig duration, p99 latency spikes) between our operator and unsafe baselines.

1) Objectives

- **O1 — Protocol-aware reconfiguration:** Implement Joint Consensus as an explicit, observable control loop with invariants and readiness gates.

- **O2 — Reproducible bugs:** Provide `make bug1` / `make bug2` scripts that deterministically reproduce safety violations on a laptop cluster.

- **O3 — Correctness evidence:** Demonstrate correctness of the safe mode under the same fault injections that break the unsafe modes. The KV store must log all operations (start time, end time, op, key, value, result) so that Porcupine can verify linearizability offline:porcupine.CheckOperations(history). The operator should assert and surface safety violations or unexpected states via events, logs, and status.
- **O4 — Performance characterization:** Quantify how reconfiguration affects the system.Metrics collected:Reconfiguration duration (AddLearner → LeaveJoint). Failover downtime (ms), leader-election latency. Throughput and p50/p95/p99 latency under load. Snapshot size, catch-up bandwidth. Tail-latency spikes during reconfig. All measurements must be automatable via make bench, using a lightweight Go load generator or vegeta.

2) Scope & Non-Goals

In scope: Raft KV Store: minimal PUT/GET interface, WAL, snapshots, durable catch-up.
Kubernetes Operator: RaftCluster CRD, controller logic, reconciliation of membership changes.
Joint Consensus Workflow: full sequence: AddLearner → CatchUp → EnterJoint → CommitNew → LeaveJoint with gating and safety checks.
Fault Injection: network partitions, leader kill, delay/loss injection using tc/iptables.
Metrics & Evaluation: throughput, latency, failover downtime, reconfiguration duration.
Correctness Checking: operation history collection + Porcupine linearizability checks.
Minimal Formal Model (optional): small TLA+/PlusCal model for membership-change invariants.

Out of scope: Sharding / Partitioning: no multi-shard coordination; single Raft group only.
Transactions: no multi-key or transactional semantics.Read Leases: follower reads or lease-based optimizations (unless time permits). Cross-Cluster Federation: single Kubernetes

cluster only. Production-Grade Security: no TLS, authN/authZ, or hardened deployment concerns.

---

3) Architecture Overview

```
+--------------------+        watches/apply        +--------------------+
|  kubectl / CRD     |  ---------------------------> |  Operator          |
|  RaftCluster CR    |                              |  (controller)      |
+--------------------+                             +---------+-----------+
                                                             |
                                                             v
                                                    +------------------+
                                                    |  StatefulSet     |
                                                    |  kv-0..kv-(N-1)  |
                                                    +------------------+
                                                    | | |
                                                    | |  +-- WAL+Snapshots (PVC)
                                                    |  +----- Raft RPCs (Headless Svc)
                                                    +-------- Admin RPCs (reconfig)
```

Discovery/Identity: Each pod receives a **stable node ID** based on its StatefulSet ordinal (`kv-0`, `kv-1`, …). A **headless Service** exposes peer-to-peer Raft RPC endpoints for log replication and leader election. Admin operations (AddLearner, Promote, EnterJoint, etc.) are served via a lightweight gRPC or HTTP endpoint.

Persistence: Each Raft pod stores state in a dedicated PVC: `wal/` — write-ahead log. `snapshots/` — periodic snapshots for fast catch-up. `metadata.json` — last-committed term/index (optional). The operator must not delete PVCs during reconfiguration to avoid losing state.

Admin plane: Minimal admin RPCs: `AddLearner`, `Promote`, `EnterJoint`, `CommitNew`, `LeaveJoint`, `TransferLeader`, `CaughtUp()`.

---

4) CRD & Status (Draft)

```
apiVersion: storage.sys/v1
kind: RaftCluster
metadata:
  name: demo
spec:
  replicas: 3
  image: ghcr.io/team/raft-kv:latest
  storageSize: 1Gi
  snapshotInterval: "30s"
  electionTimeoutMs: 300
  safetyMode: safe # [safe | unsafe-early-vote | unsafe-no-joint]
  reconfig:
    rateLimit: "1/min"
    maxParallel: 1
```

```
status:
  phase: Ready | Reconfiguring | Degraded
  leader: kv-1
  members:
    - id: kv-0
      role: voter | learner
      caughtUp: true
    - id: kv-1
      role: voter
      caughtUp: true
    - id: kv-2
      role: voter
      caughtUp: true
  jointState: none | entering | committed | leaving
  lastReconfigAt: 2025-11-08T12:00:00Z
```

spec.replicas

Target number of Raft nodes (voters + learners during transition).

spec.safetyMode

Controls whether the operator executes:

- `safe` — full Joint Consensus
- `unsafe-early-vote` — learner promoted immediately
- `unsafe-no-joint` — add/remove without joint config

spec.reconfig.rateLimit

Minimal interval between membership-change operations.

status.phase

Cluster-level health and operator progress:

- Ready — stable, not reconfiguring
- Reconfiguring — Joint Consensus in progress
- Degraded — leadership unstable or nodes missing

status.members[*].caughtUp

Learner readiness gate for promotion.

status.jointState

Directly maps to the reconfiguration workflow:

none → entering → committed → leaving → none

The operator progresses through these states in the Reconcile loop.

---

5) Control Loop (Protocol-Aware)

The operator drives Raft membership changes through an explicit, serialized workflow implemented inside the Reconcile loop:

- AddLearner → CatchUp → EnterJoint → CommitNew → LeaveJoint

All steps are reflected in status.jointState, ensuring observability and preventing re-entry.

Detecting Desired Change

When spec.replicas differs from status.members:

- If replicas increases → begin scale-out
- If replicas decreases → begin scale-in
- Only one reconfiguration may run at a time
- Enforce spec.reconfig.rateLimit before starting

Step 1: AddLearner

- Create new pod (StatefulSet handles ordinal assignment)
- Mark role as learner
- Do not allow learner to vote or serve reads/writes
- Wait for pod readiness (container health + admin RPC reachability)

Step 2: CatchUp

- Operator calls CaughtUp() on the learner
- Learner streams missing log entries and snapshots until fully caught up
- Promotion requires:

```
caughtUp == true AND durable == true
```

-

Step 3: EnterJoint

- Operator issues admin RPC:

```
POST /admin/enter-joint
```

-

- Raft commits a joint configuration entry:

  Old ∪ New

-
- Joint phase requires majority in both old and new sets

Step 4: CommitNew

- Operator issues:

  POST /admin/commit-new

-
- Raft commits the new configuration (still joint rules)
- Ensures configuration log index is monotonic

Step 5: LeaveJoint

- Operator issues:

  POST /admin/leave-joint

-
- Finalizes membership to the new set only
- If removing the leader:
  - First call TransferLeader() to a safe target
  - Then proceed with removal

Step 6: Leader Safety

- Before removing a leader pod:
  - Use a preStop hook or RPC-based leader transfer
  - Block deletion until leadership is stable
- Abort reconfiguration if leadership changes too frequently (unstable cluster)

Step 7 : Rate-Limiting & Backoff

- Enforce cooldown using spec.reconfig.rateLimit
- If any step fails or times out:
  - Mark status.phase = Degraded
  - Retry with exponential backoff
  - Do not interleave multiple reconfigs

Invariants (checked every Reconcile)

- Learners never vote
- Only one reconfig in-flight
- Promotion only after caughtUp
- Joint consensus preserved
- Leader transfer before removal
- Configuration index strictly increases

These invariants prevent unsafe ordering or concurrency during membership change.

---

6) Unsafe Baselines & Bug Reproduction

Bug 1 — Early-Vote Learner (Election Safety Violation)

Mode: `spec.safetyMode = "unsafe-early-vote"` (skip learner stage; new node becomes voter immediately).

Scenario:

1. Start {A,B,C}, commit `Put(K,1)`.
2. Scale to 4: add D as voter (log missing `K:1`).
3. Partition A|B vs C|D (netem/iptables).
4. Trigger election on C|D; D votes/wins with stale log.
5. Write `Put(K,2)` on C|D; heal partition.

Expected: linearizability violation (lost or reordered write). Porcupine should fail.

Bug 2 — No Joint Consensus (Quorum Intersection Violation)

Mode: `spec.safetyMode = "unsafe-no-joint"` (apply remove+add as independent, non-joint updates).

Scenario:

1. Start {A,B,C}.
2. Remove C and add D as two independent steps without joint phase.
3. Carefully stagger partitions so {A,B} and {B,D} can both form majorities under different configs.
4. Drive conflicting writes to the same log index.

Expected: two different commits at the same index → state machine safety violation; Porcupine fails.

Make targets (planned):

- `make bug1` → deploy unsafe-early-vote, run scripted partition, run checker.
- `make bug2` → deploy unsafe-no-joint, run scripted partition, run checker.

7) Fault Injection Tooling (local k8s)

Partition helper (pod label-based):

```
# examples/netem/partition.sh

PODS_A=(kv-0 kv-1)

PODS_B=(kv-2 kv-3)

for a in "${PODS_A[@]}"; do

  kubectl exec $a -- tc qdisc add dev eth0 root netem loss 100% to ${PODS_B[*]}

done
```

Network Partition (label-based)

Example script (partition A ↔ B):

```
# examples/netem/partition.sh

PODS_A=(kv-0 kv-1)

PODS_B=(kv-2 kv-3)



for a in "${PODS_A[@]}"; do

  for b in "${PODS_B[@]}"; do

    kubectl exec $a -- tc qdisc add dev eth0 root netem loss 100% to $b

    kubectl exec $b -- tc qdisc add dev eth0 root netem loss 100% to $a

  done

done
```

- Fully isolates two subclusters

- Deterministic for reproducing Bug1 / Bug2
- Works on any RaftCluster size (dynamic array)

Leader kill:

LEADER=$(kubectl get raftcluster demo -o jsonpath='{.status.leader}')

kubectl delete pod $LEADER

Cleanup Requirements

Each script includes a matching cleanup step:

partition.sh ↔ heal.sh

delay.sh ↔ heal.sh

The operator must not assume stable network conditions; delays and partitions are expected during evaluation.

Delay injection (optional):

kubectl exec kv-2 -- tc qdisc add dev eth0 root netem delay 200ms 50ms

All helpers will have `apply.sh`/`clear.sh` pairs to restore networking.

---

8) Correctness Plan

8.1 Design-Time Invariants

These invariants define the required safety properties of the system and guide both the operator logic and the KV store implementation:

- **Election Safety**
  Only up-to-date candidates may win elections; learners never vote.

- **Log Matching**
  Leaders are append-only; once an entry is committed, it is immutable.

- **State Machine Safety**
  No two different values may be committed at the same log index.

- **Joint Consensus**
  During reconfiguration, both the Old and New configurations must form quorums (double-majority requirement).

- **Readiness Gating**
  Promotion to voter or serving traffic is allowed **only after** durable catch-up.

- **Leader Transfer Safety**
  On leader removal or scale-in, leadership must be safely transferred before termination or isolation.

These invariants anchor the behavior of the operator's state machine and are checked at each major transition.

8.2 Runtime Checking

Linearizability (Porcupine)

- The KV client records operation history entries:

  ```
  (start_ts, end_ts, op, key, value)
  ```
-
- After each experiment, history is fed into **Porcupine** (Go library) for an offline linearizability check.

- Unsafe modes (*unsafe-early-vote*, *unsafe-no-joint*) are expected to fail under certain partitions; the safe mode should show **zero violations**.

Operator Assertions

During every Reconcile loop, the operator enforces:

- Reject promotion if `caughtUp == false`
- Allow **only one reconfiguration** at a time
- Enforce cooldown between reconfig steps
  based on `.spec.reconfig.rateLimit`
- Back off if leadership changes more than **X times within Y seconds**
- Mark cluster as `Degraded` on inconsistent or unsafe states

These guard conditions ensure the operator behaves deterministically and preserves Raft invariants.

(Optional) Small Model

A lightweight formal specification (TLA+/PlusCal or Ivy) may be included to model:

- Joint Consensus transitions
- Quorum intersection
- Invariants on configuration log entries
- Safety under concurrent add/remove operations

This model is used only for validation, not part of the runtime system.

---

9) Evaluation Plan

Environment: Local Kubernetes cluster: kind or minikube. Cluster size: 3–5 Raft replicas. Storage: local-path PVCs. Load generation: vegeta or a custom Go loadgen. All experiments scripted via make bench, make bug1, make bug2

Metrics: Throughput (ops/sec). Latency percentiles: p50 / p95 / p99. Failover downtime (ms until availability restored). Reconfiguration duration (AddLearner → LeaveJoint). Snapshot size and log size. Cold restart time. Violation rate under fault injection (Porcupine results)

Experiments:

1. **Steady-State:** throughput & tail-latency vs QPS (3 replicas). Run PUT/GET workload at increasing QPS.
2. **Failover:** delete leader under load → downtime & latency spikes.
3. **Reconfiguration:** 3→5→3 with writes → duration, throughput dip, catch-up bytes.
4. **Snapshots:** vary `snapshotInterval` → measure log size, snapshot size, and cold restart time.
5. **Correctness:**
    - Unsafe modes (bug1/bug2): violation rate vs partition length/overlap.
    - Safe mode: 0 violations across N runs (report confidence bounds).

Figures:

- Throughput vs QPS; p99 vs QPS; failover-time CDF.
- Reconfig duration CDF; timeline of throughput dip during reconfig.
- Restart time vs snapshot interval; log size vs snapshot interval.
- Violation rate curves (unsafe vs safe); one counterexample timeline.

---

10) Open Questions

- Do we adopt an existing Raft library (e.g., etcd/raft) or keep a minimal in-tree implementation?
- How much TLA+/PlusCal do we include (time-box to ≤ 1–2 days)?
- Do we add optional read leases for fast reads (stretch goal)?