POD 0006: Reconfiguration and Epoch Isolation

COMMITTED

CATEGORY	**STATUS**
Protocol Specification	Committed
AUTHORS	**CREATED**
Vikrant Rathore <vikrant@insan.ai>, Paxos Odin Contributors	2026-09-16
LAST UPDATED	**LABELS**
2026-09-17	reconfiguration, protocol, replicated-log, safety

Abstract

`paxos-odin` changes membership by deciding a **stop sign** in the same log as application commands. This record specifies how `Replicated_Log_Node` seals a configuration, how the next configuration continues on the same slot line, how a crashed host repairs an unfinished handover, and how `Log_Envelope` keeps a delayed message from one configuration out of another. It notes how the seal interacts with rotating slot ownership (POD 0010), and closes with the seeded scenarios that exercise the design and the questions left open (obligation S1, "stop-sign sealing", in `src/paxos.odin`).

Status and Implementation Boundary

This protocol is fully implemented. A pending seal clears if phase-one recovery replaces an unchosen stop-sign proposal. A decided seal persists across restarts. Under rotating ownership, concurrent owners may reach decisions above a stop slot; the wrapper prevents these unreleased choices from being delivered in the retired configuration, ensuring that the successor configuration decides those slots afresh.

Terminology and Scope

- **Configuration**: a voting membership identified by a non-zero `configuration_id: u64`. A `Replicated_Log_Node` belongs to exactly one configuration at a time.
- **Stop sign**: the log entry that names the next configuration. Deciding it is the only way a configuration ends.
- **Seal**: the state in which a configuration refuses new proposals; it begins when a stop sign is pending and persists once one is decided.
- **Handover**: the host-driven step from a sealed configuration to `replicated_log_init_from_stop` in the next one.
- **Epoch isolation**: the guarantee that a message produced in one configuration is never processed by a node in another.

In scope: everything `src/replicated_log.odin` does with stop signs, seals, and `Log_Envelope`. Out of scope: state-image transfer to a joining member and the host's storage of configuration metadata, both listed under open questions.

Stop Signs on One Slot Line

`Entry(Value, MAX_MEMBERS, MAX_METADATA_BYTES)` is a union of `Value` and `Stop_Sign(MAX_MEMBERS, MAX_METADATA_BYTES)`. A stop sign carries `configuration_id: u64`, a `small_array` of member ids, and up to `MAX_METADATA_BYTES` of opaque metadata (for example a state-image identifier). `stop_sign_init` rejects a zero configuration id (`.Invalid_Configuration_Id`), oversized metadata (`.Metadata_Too_Large`), and empty, oversized, zero, or duplicate member lists through `stop_sign_validate_members`, which is public so a wire decoder can reuse it. The input slices may alias the destination.

`replicated_log_propose_stop_sign(node, next_configuration_id, next_members, metadata, effects)` (alias `replicated_log_reconfigure`, `log_reconfigure`) proposes the stop sign through `node_propose` exactly like a command: it takes the next slot, records the proposer's own vote in its ledger and writes `Write_Vote`, and broadcasts `Accept_Message` with a pointer to that ledger value. Because commands and stop signs share the slot line, the stop sign has a definite position, and every member agrees on which commands precede it.

Sealing

`Replicated_Log_Node` keeps `configuration_id`, `stop_sign: Maybe(Stop_Sign)`, `stop_slot`, and `stop_pending`. `replicated_log_is_sealed` returns `stop_pending || stop_sign != nil`, and `replicated_log_propose`, `replicated_log_propose_batch`, and `replicated_log_propose_stop_sign` all return `.Log_Sealed` when it is true.

- **Pending.** `replicated_log_pending_stop_sign` walks the ledger's `used` bitmap (`bit_set_next`) for a voted or chosen stop sign whose id exceeds the node's own. A leader's proposal is its own vote in the ledger, so one walk covers both an acceptor's vote and a proposer's undecided proposal. `replicated_log_recalculate_stop_pending` runs after every transition that can release an entry (`replicated_log_observe_effects`) and after restore (`replicated_log_observe_durable`, which walks `chosen`). A follower that merely voted for a stop sign, or a leader that proposed one, is sealed from that moment, so this node refuses new commands while it observes the pending stop sign (`review_out_of_order_chosen_stop_blocks_proposals`).
- **Decided.** `replicated_log_observe_stop` records the earliest committed stop sign as `stop_sign` and its slot as `stop_slot`. `replicated_log_stop_sign` (alias `replicated_log_is_reconfigured`) and `replicated_log_stop_slot` expose them. The seal survives restart: `replicated_log_restore` and `replicated_log_restore_learner` call `replicated_log_observe_durable` (`review_stop_seal_restore_and_completed_history`).

A pending stop sign alone does not authorise a handover. The `.Log_Sealed` hint says so: the host must finish deciding and delivering the stop sign, then install the agreed state in its next configuration.

Crash Repair

If the host crashes between proposing a stop sign and starting the next configuration, it restores with `replicated_log_restore` and asks `replicated_log_pending_stop_sign(node)`. A decided stop sign is returned first; otherwise the accepted stop sign restored from the journal is returned; a leader proposal survives a crash only through its durable local vote. The host can then wait for the decision, or campaign again so that phase one re-proposes the highest-ballot vote (which may be the stop sign), and it never has to guess whether a handover was under way (`review_log_pending_stop_replaced_during_replay`).

Continuing in the Next Configuration

`replicated_log_init_from_stop(node, id, stop, stop_slot, anchor, options)` builds a `Membership` from the stop sign's members, checks that the stop sign's id is non-zero, and calls

`replicated_log_continue_at(node, id, stop.configuration_id, membership, stop_slot, anchor, options)`. That in turn calls `node_continue_at`, which starts an empty window with `memory_floor = delivered_through = stop_slot`, `next_slot = stop_slot + 1`, and `ledger.anchor = anchor` (rejecting `anchor.chosen_trim_slot > stop_slot` with `.Trim_Regression`). A node whose id is not among the new members gets `.Not_Member` from the local-id check in `node_init`; the handover scenarios check that a removed voter is refused.

Slot numbers are therefore global across configurations: the first command of configuration 2 lands in `stop_slot + 1`. The host-certified continuation floor is `stop_slot`, so phase one starts above sealed history. An inherited trim anchor may certify a shorter prefix; it does not substitute for the required state handover.

Configuration ids increase strictly. `replicated_log_propose_stop_sign` returns `.Configuration_Id_Regression` when `next_configuration_id <= node.configuration_id`, and `replicated_log_next_stop` ignores any stop sign whose id is not greater than the node's own, so a replayed older stop sign cannot reseal a newer configuration.

Epoch Isolation on the Wire

A bare `Envelope(Entry(...))` carries `from`, `to`, and the message, and nothing that says which configuration produced it. If a transport can deliver a message late, across a handover, a node in configuration 2 could receive an `Accept_Message` or `Prepare_Message` from configuration 1 whose ballot and slot look valid in its own window. `replicated_log_step` therefore exists in two forms:

- `replicated_log_step(node, envelope, effects)` processes a bare envelope. Its doc comment restricts it to transports that already isolate configurations, for example by opening a fresh authenticated channel per configuration and closing the old one before the handover completes.
- `replicated_log_step_checked(node, message: Log_Envelope, effects)` compares `message.configuration_id` with `node.configuration_id` before the core inspects anything. On a mismatch it calls `effects_reset(effects)` and returns `.Configuration_Mismatch`: no writes, no messages, no state change. `replicated_log_envelope(node, envelope)` stamps each outbound envelope with the node's configuration id. Both are reachable through the `step` and `log_step` proc groups; the argument type selects the checked path.

`Log_Envelope(Value, MAX_MEMBERS, MAX_METADATA_BYTES)` is the wire type hosts should serialise. The `.Configuration_Mismatch` hint is explicit: discard the stale message or route it to its original configuration; never relabel old traffic with the new id. The inner envelope's value is a pointer into the sender's ledger (POD 0003), so a transport that queues `Log_Envelope`s copies the `Entry` at enqueue exactly as the core `Packet` idiom does; `Seal_Packet` in `tests/test_reconfiguration_sim.odin`, which pairs a configuration id with a `Packet(Seal_Entry)`, is that copy.

Stop Signs under Rotating Ownership

A stop sign is an ordinary decree: `Entry` is a union of `Value` and `Stop_Sign`, and nothing in the core knows which variant a cell holds. With `Node_Options{rotating_ownership = true}` (POD 0010), `replicated_log_propose_stop_sign` therefore lands the stop sign wherever `node_propose` lands any value: in the proposer's next own slot, at the round-zero ballot `ownership_ballot(node.id)`, with no phase one. The proposer of a stop sign is its slot's owner. Sealing works as above on every node that votes for the stop sign or sees it chosen, and `replicated_log_init_from_stop` continues at `stop_slot + 1` as before.

Two consequences follow from the per-slot structure, and one rule in the wrapper answers both:

- Another owner that has not yet heard of the stop sign can still suggest in its own slots above `stop_slot`, because the seal is per node and begins only when the node observes the stop sign. Such a suggestion

can be chosen by the old configuration's quorums, and the no-op skips of `tick_ownership` (which run inside `node_tick`, not through `replicated_log_propose`) can be chosen there too.

- **Decisions above a decided stop sign are abandoned.** `replicated_log_observe_effects` cuts every released entry above `stop_slot` out of the batch (`replicated_log_abandon_above_seal`), `replicated_log_read` and `replicated_log_read_decided` report those slots as undecided, and `replicated_log_decided_through` never exceeds `stop_slot`. The next configuration decides those slots afresh with its own quorums. This is the rule of Lamport, Malkhi, and Zhou: the acceptors of an instance are those of the configuration that owns it, so a value chosen at slot $s + 1$ by C_1's acceptors is not a choice under C_2, and nothing that was never released can have been applied. A command abandoned this way was never acknowledged; its client retries it, as it would after any timeout, and the command's id deduplicates it.

The scenario `reconfiguration_sim_ownership_abandons_decisions_above_the_seal` exercises exactly this: owner 2 seals in slot 2 while owners 3 and 1 get slots 3 and 4 decided; the test requires that some node's ledger holds slot 3 as chosen, that no node releases or reads it, and that the next configuration decides slot 3 with a new command.

Validation and Acceptance Gates

`tests/test_reconfiguration_sim.odin` runs four scenarios on a three-voter `Replicated_Log_Node(u64, 4, 8, 2, 32)`, each over 16 seeds. The host commit sequence in the harness carries an oracle: a node sealed by a decided stop sign never releases an entry above it. Delivery order is shuffled per seed, every journal persists before any message leaves, and every message crosses the network as a `Log_Envelope`.

1. `reconfiguration_sim_seal_survives_drop_duplicate_and_reorder`: a command and the stop sign race for adjacent slots; the accept that carries the stop sign is dropped once and duplicated once.
2. `reconfiguration_sim_membership_handover_reaches_new_configuration`: members `{1, 2, 3}` hand over to `{2, 3, 4}`; node 1 is refused by `log_init_from_stop`.
3. `reconfiguration_sim_one_for_one_voter_replacement`: `{1, 2, 3}` becomes `{1, 2, 4}` starting from configuration 7, with a different leader.
4. `reconfiguration_sim_ownership_abandons_decisions_above_the_seal`: under rotating ownership, owner 2 seals in its own slot while owners 3 and 1 get later slots decided; the decisions above the seal are abandoned and the next configuration re-decides them.

Every scenario checks the same four oracles: `seal_expect_agreement` (every member decided the same stop sign in the same slot, with identical prefixes entry by entry, and `decided_through` equals the stop slot); `seal_expect_nothing_after` (no slot past the seal is decided and `propose` returns `.Log_Sealed`); `seal_expect_replay_keeps_seal` (replaying each journal into a fresh `Ledger` with `ledger_replay_fold` and calling `restore` rediscovers the seal and its slot); and `seal_expect_next_epoch_decides` (after `log_init_from_stop`, a new leader decides three commands at `stop_slot + 1..3` and every member reports the new `log_configuration_id`).

`reconfiguration_cluster_handover_rejects_old_epoch` in `tests/test_reconfiguration.odin` additionally holds back one real accept from configuration 10, completes the handover to configuration 11, and asserts that the checked `log_step` returns `.Configuration_Mismatch` with zero writes and zero messages before the new configuration goes on to decide.

Open Questions

1. **Joiner catch-up.** A new member named by a stop sign starts from `log_init_from_stop` with an empty window and an anchor at `stop_slot`. It has none of the sealed history and no application state. The library assumes the host installs the state image named by the stop sign's metadata before the node

serves reads; there is no protocol step that transfers it, and `Serve_Range_Request` only covers slots the serving node once held.

2. **Trimming across handovers.** The next configuration inherits one `Trim_Anchor` whose `chosen_trim_slot` is the stop slot. Whether the old configuration's later trim ids may be reused, and how a host that restores already trimmed stop history recovers its configuration metadata, is left to the host (POD 0007 notes the second limit). A future record should fix the relationship between `trim_id` sequences and configuration ids.
3. **Learner handover.** `replicated_log_init_learner` follows one configuration id. A learner that observes a decided stop sign (`review_log_learner_observes_stop_and_catchup`) still needs the host to re-initialise it for the next configuration.
4. **Traffic after the seal under rotating ownership.** A sealed node still ticks: it keeps skipping and retransmitting in its own slots above the seal until the host retires it, and those decisions are abandoned. Holding skips back once a node observes the seal would save messages during the handover; it is not needed for safety.

References

- Lamport, Leslie, Malkhi, Dahlia, and Zhou, Lidong. "Reconfiguring a State Machine." ACM SIGACT News, 2010.
- Lamport, Leslie. "The Part-Time Parliament." ACM TOCS, 1998.
- POD 0002: Paxos-Odin: Architecture and Pure State Machine Design.
- POD 0003: Durability Contracts, Window Reuse, and Trim Anchors.
- POD 0005: The Idiomatic Odin API Surface.
- POD 0010: Rotating Slot Ownership.