POD 0003: Durability Contracts, Window Reuse, and Trim Anchors

COMMITTED

CATEGORY
Protocol Specification

STATUS
Committed

AUTHORS
Vikrant Rathore <vikrant@insan.ai>, Paxos Odin Contributors

CREATED
2026-09-16

LAST UPDATED
2026-09-17

LABELS
durability, storage, protocol

Abstract

This specification states the durability obligations a host of `paxos-odin 0.2.0` must meet, the runtime gate that catches the most common violation, the five durable records and the copy obligation that follows from their pointer payloads, the pre-durable rule and its round-zero exception, the memory floor that lets a bounded window be reused, the trim anchors that let an acceptor answer for a prefix it no longer holds, and the journal replay contract. Every rule below is enforced or documented in `src/ledger.odin`, `src/effects.odin`, `src/node.odin`, and `src/consensus.odin`.

Status and Implementation Boundary

The durability contract is implemented. Recovery selection is now frozen before phase two and retained across backpressure retries (POD 0009). This does not extend an effect pointer's lifetime: a host must copy queued payloads before the next transition, including independently owned copies of duplicated simulator packets.

An error does not generally imply an empty effects batch or an unchanged node. The host must inspect and finish the batch before deciding how to handle the error. Ownership batch admission is a specific stronger contract: its read-only `own_slot_probe` preflight rejects an unavailable batch without writing a vote or moving the frontier. The Python bridge implemented in POD 0011 preserves both rules.

The Core Durability Contract

Paxos survives crashes only if a promise or a vote that a peer may have observed is never forgotten (obligation D1, "indelible ink", in `src/paxos.odin`).

> **The Durability Rule:**
> *Persist and sync every record in `Effects.writes`, call `confirm_writes_durable`, and only then transmit any `Effects.messages` from the same transition.*

If a node sends a `Promise_Message` or an `Accepted_Message` before the matching `Write_Promise`, `Write_Promise_At`, or `Write_Vote` is on stable storage, a crash can revert the promise; after restart the node may promise or vote differently, and two different values can be chosen for one slot.

The Five Durable Records

`Write(Value)` in `src/ledger.odin` is a union of five records. The table names the procedure that emits each one.

Record	Meaning	Emitted by
`Write_Promise{ballot}`	The global promise (`Ledger.promised`, Lamport's `maxBal` for every decree at or above the recovery base).	`start_campaign` for its own ballot; `on_prepare` upon receiving a `.Global` prepare; `on_heartbeat` when adopting a heartbeat ballot above the current promise.
`Write_Promise_At{ballot, slot}`	A promise for one decree only (`Ledger.promised_at[cell]`), made to a `.Bounded` prepare, which is a revocation under rotating ownership.	`promise_bounded`, once per decree in `[first, last]` above the memory floor.
`Write_Vote(V){ballot, slot, value: ^V}`	A vote (`vote_ballot[cell]`, `value[cell]`, state `.Voted`).	`send_accept` for the proposer's own vote; `on_accept` for an acceptor's vote.
`Write_Chosen(V){slot, value: ^V}`	A decision (state `.Chosen`). Derived state: a decision is implied by a write quorum of votes.	`record_commit`, on a local quorum, a `Commit_Message`, a recovered decision, or a host-certified value.
`Write_Trim`	The trim anchor (`Ledger.anchor`), an alias of `Trim_Anchor`.	`node_install_chosen_trim`. `node_begin_recovery` applies the anchor through `ledger_apply` and relies on the host to persist the image and anchor it installed.

`effects_requires_power_loss_barrier` returns true only when the batch holds a `Write_Promise`, `Write_Promise_At`, or `Write_Vote`; `Write_Chosen` and `Write_Trim` are derived state a host may persist with a cheaper barrier (`test_effects_power_loss_barrier_flag`).

Pointer Validity and the Journal Copy

`Write_Vote` and `Write_Chosen` carry `value: ^Value`, not a value. The pointer refers into the ledger of the node that produced the record (`&l.value[cell]`), or into `node.pass_through` for a decision released past the window edge, and it is valid until the next transition on that node. The same rule covers `Committed.value` and the `value` field of `Promise_Message`, `Accept_Message`, and `Commit_Message`.

Two obligations follow:

1. **Copy at the journal.** A host persists every write before it runs another transition on the node, so serialising the record inside the commit sequence is enough. A host retaining records in memory (such as a test journal or an in-process queue) must copy the value during append. `Journal_Record{write, value}` with `journal_append` in `tests/harness.odin`, and `Sim_Record` with `persist_sim_write` in `sim/simulation.odin`, demonstrate this reference pattern: the record is stored with its own copy, and `journal_replay` (or `sim_restart_node`) points `x.value` back at that copy before folding the record into a fresh `Ledger`.
2. **Copy at the transport.** An envelope points into the sender's ledger, which the sender may overwrite in its next transition. `Packet{envelope, value}` with `packet_of` (copy on enqueue, using `message_value`) and `packet_envelope` (repoint at the packet's copy for the duration of `step`) appears in `tests/harness.odin`, `examples/counter.odin`, `sim/simulation.odin`, and `bench/main.odin`. A real codec does the same thing by serialising before the next transition.

POD 0009 records why the payloads are pointers.

The Runtime Durability Gate

`Durability_Gate` is an enum with two values and is the fifth type parameter of both `Node` and `Effects`.

`.Enforced` (the default)

`Effects` carries one flag, `writes_pending`, which `effects_add_write` sets. With `GATE == .Enforced`:

- `effects_messages_slice` while `writes_pending` is true calls `host_order_violation("messages_slice before confirm_writes_durable")`;
- `effects_reset` while `writes_pending` is true calls `host_order_violation("reset discarded unconfirmed writes")`.

`host_order_violation` prints to standard error and exits the process with status 1:

```
-- DURABILITY ORDER VIOLATION ------------------------------------------------

messages_slice before confirm_writes_durable.

Hint: Persist and sync the pending writes before calling confirm_writes_durable(),
then transmit messages and reset the batch. Never confirm a failed write.
Recover from the durable journal before restarting this stopped node.
```

The second diagnostic reads `reset discarded unconfirmed writes.` under the same banner and hint. `tools/check_contracts.py` builds both misuse programs in `-debug` and `-o:speed` and requires the named fragment and `Hint:` in the output; it also builds the correct ordering and a zero-value batch and requires both to exit cleanly (four fixtures, eight runs).

`.Host_Managed`

`Node(Value, M, W, C, .Host_Managed)` and its matching `Effects` compile the two checks away. The doc comment on `Durability_Gate` names the four rules such a host must guarantee by construction:

1. every write of a transition is durable before any message of that transition reaches a peer;
2. a batch is never discarded while it still holds unconfirmed writes;
3. committed entries are applied only after their commit record is durable;
4. a crash between the writes and the barrier is recovered from the journal, never by confirming writes that did not complete.

This mode exists for hosts that group several transitions behind one storage barrier. `test_host_managed_gate` verifies that this configuration compiles and returns messages even with an unconfirmed promise in the batch.

Batch lifecycle

- The zero value of `Effects` is ready: `writes_pending` is false and every list is empty (`test_zero_value_effects_are_ready`, and the `zero_value_is_ready` fixture).
- `effects_init` clears the batch without consulting the gate. It is for fresh memory and for a batch that a crashed process can never complete; the simulator calls `paxos.init(effects)` after a simulated crash instead of falsely confirming.
- `effects_reset` clears with the check, and every protocol transition calls it first.

The Pre-Durable Rule and the Round-Zero Exception

`effects_pre_durable_messages` returns a `Pre_Durable_Iterator`; `pre_durable_next` yields only `Accept_Message` envelopes, and only those whose ballot has `ballot_round > 0`. Such an accept asks peers to persist a vote and claims nothing about the sender's own durability; a restarted proposer always campaigns at a fresh round, so no later message can be confused with the one that left early (`test_pre_durable_messages_iterator` checks that heartbeats and commits are held back).

An owner's round-zero suggestion under rotating ownership is the exception. `ownership_ballot(owner) = ballot_make(0, 0, owner)` is the same ballot after every restart: there is no fresh round to campaign at. The owner's own `Write_Vote` is therefore the only durable record that the instance was used. If the accept left before that vote was durable and the owner crashed, the restarted owner could suggest a different value in the same slot under the same ballot, and one ballot would carry two values. The simulator's vote oracle (`persist_sim_write`, "ballot accepted two values in slot") found exactly this under `--ownership` with pre-durable delivery; the rule in `pre_durable_next` is the fix, and POD 0007 records the finding.

Memory Floor and Window Reuse

Slots are 64-bit and never wrap; the window holds `WINDOW_SLOTS` cells, a power of two, indexed by `cell_of(slot) = (slot - 1) & (WINDOW_SLOTS - 1)`.

- `node_advance_memory_floor(node, through)` records that the host has durably consumed every released entry through `through`. It returns `.Invalid_Slot` when `through > delivered_through` (the value `decided_through` reports), so the floor never passes what was released, and it never moves backwards.
- `node_propose` returns `.Window_Full` when `next_slot - memory_floor > WINDOW_SLOTS`; `node_propose_batch` refuses the whole batch when it would not fit (`review_batch_backpressure_is_atomic`). Under ownership `next_usable_own_slot` applies the same bound to the node's next own slot and `own_slots_available` to a batch. The `.Window_Full` hint tells the host to deliver, consume, and advance the floor, never to advance past the released prefix.
- `claim_live(node, slot)` in `src/consensus.odin` is the only live path that reuses a cell. It refuses `slot <= memory_floor`, returns the cell if `ledger.slot[cell]` already holds `slot`, and otherwise calls `ledger_open` only when the cell is empty or holds an older slot that is both at or below the floor and `.Chosen`. An open vote is never overwritten (`review_flexible_quorums_and_window_reuse`).
- `record_commit` for the slot just past the window edge (`slot == delivered_through + 1` when no cell can be claimed) releases the decision through `node.pass_through` so one transition can free the window; this is why `committed` has `WINDOW_SLOTS + 1` entries.
- `resolve_chunk` bounds recovery proposals by `memory_floor + WINDOW_SLOTS` so a new leader cannot outrun its own window, and returns false so a tick retries the rest.
- `on_learn` answers a `Learn_Message` from the `chosen` bitmap and, for any requested slot at or below the floor, emits `Serve_Range_Request{peer, first, count}` in `Effects.requests`; the host serves that range from its journal or state image (the simulator replays its applied values as `Commit_Message`s).

Trim Anchors

`Trim_Anchor{trim_id: u64, chosen_trim_slot: Slot}` certifies that every slot at or below `chosen_trim_slot` is chosen and has been folded into a host state image identified by `trim_id`. The anchor carries no hash: the core compares anchors by identity, and the host binds the image's checksum to the id itself. It is durable state (`Ledger.anchor`) and travels in `Promise_Range_Message.anchor`.

- `node_install_chosen_trim(node, anchor, effects)` adopts a host-certified anchor. It rejects `chosen_trim_slot > delivered_through` with `.Invalid_Slot`, a lower `trim_id` or lower slot with `.Trim_Regression`, and an equal `trim_id` that differs in its slot with `.Trim_Regression` (`review_live_trim_rejects_conflicting_identity`). On success it emits `Write_Trim` and raises `memory_floor` to `min(chosen_trim_slot, delivered_through)`.
- `on_prepare` skips cells at or below the anchor and reports the anchor in `Promise_Range_Message`; `on_accept` and `record_commit` ignore slots at or below it. `quorum_fences` takes the maximum anchor and `chosen_through` across promises, so a new leader starts proposing above everything any quorum member certified (`review_recovery_preserves_fences_across_chunks`).
- `node_begin_recovery(node, anchor)` installs a certified image: it applies `Write_Trim` through `ledger_apply`, clears only cells at or below the anchor that hold an open vote, keeps every vote above it (they may belong to a chosen quorum; `review_snapshot_preserves_votes_above_anchor`), resets volatile leader state, and returns to `.Follower`.
- `node_continue_at(node, id, membership, floor, anchor)` starts an empty node whose window resumes at `floor + 1` with an inherited anchor; it rejects `anchor.chosen_trim_slot > floor` with `.Trim_Regression`. `replicated_log_init_from_stop` uses it across a configuration handover (POD 0006).

Journal Replay Contract

Two procedures in `src/ledger.odin` apply a `Write` to a `Ledger`. Both read the value through the record's pointer, so the host points each record at its journaled copy before the call.

`ledger_apply` (strict)

For a journal written in one process lifetime, in order:

- `Write_Promise` below `promised` is `.Promise_Regression`; otherwise it replaces `promised`.
- `Write_Promise_At` claims the cell (`.Window_Overrun` if an earlier, open, uncertified slot still holds it); a ballot below `promised_at[cell]` is `.Promise_Regression`; otherwise it replaces the per-decree promise.
- `Write_Vote` below `promised` is `.Promise_Regression`; after claiming the cell, below `promised_at[cell]` is `.Promise_Regression`; one ballot with two values is `.Conflicting_Value`; a value that differs from a decision already in the cell is `.Conflicting_Commit`. Otherwise `promised_at[cell]` becomes the vote's ballot (a vote is a promise for its decree) and the vote is recorded unless the cell is already `.Chosen`.
- `Write_Chosen` whose cell a later slot already owns returns `.None` and is skipped (derived state); a different value in a `.Chosen` cell is `.Conflicting_Commit`; otherwise the decision is recorded.
- `Write_Trim` with a lower id or slot, or the same non-zero id with a different slot, is `.Trim_Regression`.

`ledger_claim` (replay's cell claim) may reuse a cell whose old slot is `.Chosen` or lies at or below the durable anchor, even when the cell holds only a vote (`review_replay_reuses_certified_trimmed_vote`).

`ledger_replay_fold` (lifetime journals)

For a journal appended across restarts, in which promises and votes legitimately appear out of monotone order:

- `Write_Promise` folds to the maximum seen.
- `Write_Promise_At` and `Write_Vote` are skipped (`.None`) when a later slot already owns the cell and are `.Window_Overrun` only when an earlier slot does; the per-decree promise folds to the maximum; a

vote still fails with `.Conflicting_Value` when one ballot carries two values, and is not recorded over a decision.
- `Write_Chosen` and `Write_Trim` delegate to `ledger_apply`.

The simulator's `sim_restart_node`, `journal_replay` in `tests/harness.odin`, and the reconfiguration scenarios replay with it, then call `node_restore(node, id, membership, ledger, floor, options)`, which clears cells at or below `max(floor, anchor.chosen_trim_slot)` that hold only an open vote and recomputes `next_slot`, `leader_base`, `highest_seen`, and (under ownership) `own_next` from the ledger.

Hosts that write one journal per process lifetime use `ledger_apply`; hosts that append across restarts use `ledger_replay_fold`. Neither procedure allocates.

Validation and Acceptance Gates

- `test_effects_power_loss_barrier_flag`, `test_pre_durable_messages_iterator`, `test_host_managed_gate`, and `test_zero_value_effects_are_ready` in `tests/test_durability.odin`.
- The four durability fixtures in `tools/check_contracts.py`, each built in both profiles.
- `test_node_restore_and_recovery`, `review_replay_reuses_certified_trimmed_vote`, `review_snapshot_preserves_votes_above_anchor`, and `review_live_trim_rejects_conflicting_identity`.
- The seeded simulator persists every write through an oracle that rejects promise regression and votes below the promise, records a decision as soon as a durable write quorum exists (before any leader announces it), crashes at `Before_Writes`, `Partial_Writes`, and `Partial_Messages`, advances the memory floor only half of the time so full-window paths are exercised, serves evicted history from the host image, and runs in both the single-leader and the ownership mode.

The Python Bridge as an Audited Host

POD 0011's bridge compiles the core with `.Host_Managed` and enforces the four obligations itself. The reason is specific: `host_order_violation` calls `os.exit`, and a host running inside a Python interpreter cannot accept a process kill with no traceback, no unwound `finally` block and no chance to close its journal. Returning a status is the only behaviour a hosted caller can act on.

The obligations are met as follows. Every output accessor requires the batch to be confirmed, so no message, released entry or served range can be read before its writes are durable. No transition may begin while a batch is unfinished, and finishing requires confirmation, so a batch is never discarded while it holds unconfirmed writes. Closing a node with an unconfirmed batch is permitted - since `close` must execute reliably within a `finally` block - but it reports the count of abandoned records rather than hiding them. Recovery is journal replay; the bridge never confirms writes whose persistence is uncertain.

A second library, compiled with `.Enforced`, runs the entire Python test suite as a standing proof that the bridge never trips the core's own gate. Both libraries must return identical status for the same trace, and a test asserts it.

References

- Lamport, Leslie. "The Part-Time Parliament." ACM TOCS, 1998.
- Lamport, Leslie. "Paxos Made Simple." ACM SIGACT News, 2001.
- Lamport, Leslie, Malkhi, Dahlia, and Zhou, Lidong. "Reconfiguring a State Machine." ACM SIGACT News, 2010.
- POD 0002: Paxos-Odin: Architecture and Pure State Machine Design.
- POD 0006: Reconfiguration and Epoch Isolation.
- POD 0009: The Data-Oriented Ledger.

- POD 0010: Rotating Slot Ownership.