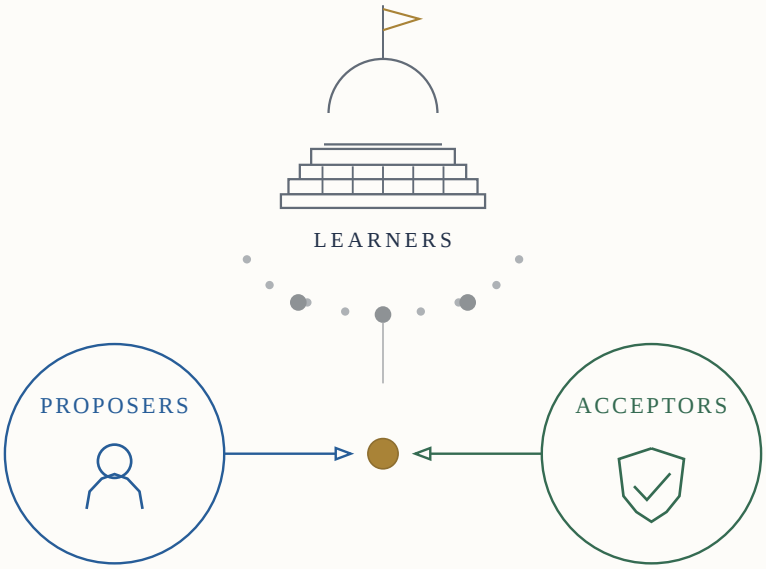
PAXOS-ODIN

THE PART-TIME PARLIAMENT IN ODIN

A LITERATE GUIDE TO CONSENSUS
FOR UNRELIABLE TIMES



$$|Q_1| + |Q_2| > N \quad \Rightarrow \quad Q_1 \cap Q_2 \neq \varnothing$$

Agreement, even when members come and go.

VIKRANT VARMA

AND THE PAXOS-ODIN COMMUNITY

About this book

The public monorepo is insanai/paxos-odin. The project website provides this book and the PODs as HTML, their PDFs, and the generated Python API reference.

This book explains one fundamental algorithm: Leslie Lamport's Paxos consensus protocol. It also explains one concrete, bounded implementation of that algorithm written in the **Odin** programming language: `paxos-odin`.

The two tasks are kept together throughout this text. A line of systems code is easier to trust when we understand the mathematical proof obligation that requires it. Conversely, a proof of safety is easier to remember when we can point directly to the struct field that carries its meaning.

The source is written in Typst. The diagrams use Fletcher and CeTZ. The code is idiomatic Odin. The consensus protocol is grounded in Leslie Lamport's seminal papers, "The Part-Time Parliament" (ACM TOCS 1998) and "Paxos Made Simple" (2001); the reconfiguration and epoch-sealing layer is derived from the stop-sign construction described in Lamport, Malkhi, and Zhou's "Reconfiguring a State Machine".

> *Early in this millennium, the Aegean island of Paxos was a thriving mercantile center. Though its citizens were very busy with their business, they needed a government to lead them and make communal decisions.*
>
> - Leslie Lamport, "The Part-Time Parliament"

The primary promise

A careful reader should be able to derive the core Paxos safety invariant, trace it directly into the struct fields and effects of `paxos-odin`, run a replicated three-node counter in memory, build an event-driven host application, and inspect the safety argument and its assumptions without confusing a consensus protocol contract with an application-level contract.

Preface

The usual introduction to Paxos starts too late. It begins with "Prepare" and "Accept" messages. Those messages then look like arbitrary rules from a game whose underlying purpose was forgotten.
We shall start much earlier. We shall ask a simple question:
Three machines must write one value on three pieces of paper. A machine may stop at any moment. A network messenger may vanish into thin air. No machine may ever erase ink that was once written. How can the machines guarantee that two different values are never declared final?
We build the answer by testing small examples. Each example exposes a flaw in a tempting solution and gives us a condition the next solution must satisfy. Prepare and Accept then become messages that carry those conditions between machines.
We compute small traces, predict outcomes before seeing answers, and explain each transition in plain language. Then we move a crash or delay a message and ask which facts still hold. The formal argument collects those facts into invariants.
The implementation follows the exact same pedagogical order:

1. First come values, node identities, and ballots.
2. Next come promises, durable ballots, and votes.
3. Then come multi-slot logs, leader elections, log hole filling, and stable storage, and the chapter that proves the whole construction safe.
4. Finally come bounded memory windows, rotating slot ownership, stop signs, and state machine replication.

At each step, we ask two fundamental questions:

- **What can go wrong?**
- **Which invariant prevents it?**

0.1 How to read the book

Parts I through III derive the protocol from one safety question, end with a sequence of decisions, and close with the safety argument: axioms, lemmas, and the proof that each departure from the textbook preserves agreement. Part IV is the library: the bounded state machine, the host contract, the advanced features, rotating slot ownership, and the coding rules that keep the code reviewable. Part V builds three systems on it. Part VI is the evidence and its limits. Part VII is the desk reference, with the answers to selected exercises, and Part VIII maps Lamport's paper to the procedures that implement it. Part IX develops the proposed Python SDK, keeping its future interface distinct from the implemented Odin core.

0.2 Audience and prerequisites

You need sets, integer arithmetic, and the willingness to accept that a process can stop between any two instructions. Odin is read, not required: every excerpt is short and explained. If you already know Paxos, start with the checkpoint at the top of Part I and skip forward when it passes.

0.3 Notation

- N voters; a quorum Q; Q_1 the phase-one (read) quorum and Q_2 the phase-two (write) quorum. Intersection is written $Q_1 \cap Q_2 \neq \varnothing$.
- A ballot is the triple (r, p, n): round, priority, node, ordered lexicographically. The library packs the triple into one 64-bit integer so that the lexicographic order is integer comparison.
- The owner of slot s under rotating ownership is member $(s-1) \bmod N$ in ascending node-id order; its ballot in that slot has round 0.
- Slots are one-based; $s = 0$ means "no slot".
- Code identifiers appear in `monospace`; Odin types are `Ada_Case`, procedures are `snake_case`, error values are written with a leading dot, as in `.Not_Leader`.

0.4 Commands used in the book

shell

```
make build                      # library object, simulator, benchmark, CLI into bin/
make test                       # odin test tests
make check                      # style, tests in two builds, contracts, 240 seeded
simulations, smoke runs
make example                    # the three-node counter
./bin/paxos-sim --seed=7 --steps=10000 --nodes=5 --verbose
./bin/paxos-sim --seed=7 --steps=10000 --nodes=5 --ownership   # every node proposes
./bin/paxos-bench --durable     # in-memory modes plus journal-and-fsync modes
make bench-matched              # matched CPU workloads across four libraries
make bench-profile              # Callgrind and Massif evidence
make bench-compare              # historical harness, including durable modes
make docs                       # this book and the POD records as PDF
```

0.5 Accompanying artefacts

- The library in `src/`: the core in ten files (`ballot.odin`, `bit_set.odin`, `membership.odin`, `ledger.odin`, `messages.odin`, `effects.odin`, `node.odin`, `election.odin`, `consensus.odin`, `ownership.odin`), then `replicated_log.odin` (stop signs and configuration-checked envelopes), `learner.odin`, `errors.odin`, and `paxos.odin` (the unified surface).
- `examples/counter.odin`: the three-node replicated counter walked through in Part V.
- `sim/`: the seeded fault simulator with its oracles, in single-leader and rotating-ownership modes.
- `bench/`: the in-memory and durable benchmarks.
- `tools/check.py`: the complete verification run.

> **A principle of verification**
> Testing can reveal a broken invariant. It cannot create an invariant. We first state the formal mathematical reason that the consensus engine is safe. We then use deterministic simulation and unit tests to search for flaws in our code and our understanding.

Contents

Pedagogical Approach and Structure

Distributed consensus is notoriously difficult because one must simultaneously account for process crashes, network partitions, message loss, arbitrary delays, competing leaders, and non-volatile storage invariants. Verbose descriptions rarely help; rigorous structure and progressive disclosure do. This book builds understanding in deliberate layers, repeatedly anchored to one central question.

> **The Central Question**
> What fundamental invariant prevents two distinct values from ever being chosen in the same slot?

Our editorial approach is guided by four core principles, directly reflecting the architectural values articulated in POD 0001:

1. **Visualize concrete scenarios first.** We begin with three voters, one value, and a single slot before introducing generalized quorums and multi-slot logs. We describe the physical situation in clear language before introducing formal symbols.
2. **Make the reasoning inspectable.** We state explicit assumptions, precisely define terms such as "chosen", and formulate safety invariants before presenting the state transitions that preserve them. Safety guarantees are kept strictly distinct from liveness conditions.
3. **Treat code as executable specification.** We place Odin implementation excerpts directly beside the mathematical proof obligations they satisfy. We trace entire message flows whenever distributed interactions become subtle.
4. **Exercise mechanical sympathy.** We assign each variable a single unambiguous meaning, cleanly distinguish buffer indices from consensus slots, and use targeted counterexamples to demonstrate why tempting shortcuts fail.

Diagrams are designed to perform explanatory work: clarifying which participant witnessed an earlier vote, which event must complete before sending a reply, and when storage may be safely reused. Labels convey essential semantic ordering, while colour serves as a reinforcing cue.

0.6 The Three Levels of Understanding

Every consensus mechanism in this book is examined at three distinct levels:

Level	Core Question	Evidence of Mastery
1. Safety Invariant	What must never happen?	You can articulate the invariant clearly in plain language.
2. State Transition	Which state change preserves the invariant?	You can trace protocol events and verify that no past commitments are violated.
3. Odin Implementation	Which struct field, durable write, and message?	You can navigate `Node` and `Effects` while maintaining the strict persist-before-send contract.

Placing Odin excerpts directly alongside theoretical invariants is intentional: the code is the proof obligation made concrete, and the invariant provides the exact rationale for why the code is structured as it is.

0.7 Chapter Structure

To ensure concepts are internalized and verifiable, instructional chapters follow a structured progression:

1. **Objectives & Prerequisites:** Clear statements of concepts and skills introduced in the chapter.
2. **Thought Experiment:** An initial prediction exercise that highlights subtle failure modes before presenting the solution.
3. **Worked Derivation:** Step-by-step analysis of messages, state mutations, and underlying justifications.

4. **Code Inspection:** Concrete Odin procedures and data structures implementing the mechanism.
5. **Review & Exercises:** Structured questions and failure variations to test understanding.

0.8 Suggested Reading Pathways

Focus	Recommended Sequence	Practical Verification
Protocol Engineer	Parts I-III (Foundations, Single-Decree, Multi-Paxos, and Safety Argument), then Part VII (Reference), followed by Parts IV-VI.	Diagram quorum intersections from memory; complete the protocol exercises before inspecting solutions.
Systems Implementer	This introduction, followed by Parts IV-VI (Library Architecture, Applications, and Evidence), returning to Parts I-III when protocol rationales are needed.	Run `make check`; trace transitions in the replicated counter example; inspect simulation assertions under injected network and crash faults.

0.9 Repository Architecture

The principles and claims in this book correspond directly to runnable software in the repository:

- `src/`: The pure consensus engine. A bounded Multi-Paxos `Node` whose state resides in a columnar `Ledger`; a caller-owned `Effects` batch; optional rotating slot ownership; a `Replicated_Log_Node` supporting epoch isolation via stop-sign reconfiguration; a non-voting `Learner`; and an `Error` enum with explanatory diagnostic hints for every condition.
- `examples/counter.odin`: A standalone three-node replicated counter illustrating the complete host integration loop in approximately 130 lines of code.
- `tests/`: 79 deterministic test suites, including an exhaustive election test matrix, seeded reconfiguration scenarios, and rotating slot ownership tests.
- `sim/`: A deterministic fault simulator verifying safety invariants (agreement, validity, monotonic commitments, contiguity, and liveness) across single-leader and multi-proposer rotating configurations.
- `bench/`: Matched CPU benchmarks comparing Odin against Zig, Rust (OmniPaxos), and C (LibPaxos3), accompanied by callgrind and memory profile datasets.
- `tools/check.py`: Complete verification harness covering code style, multi-build compilation, durability contracts, and extensive simulation runs.

0.10 Prerequisite Self-Assessment

Before proceeding to Part I, test your intuition against these four foundational questions:

1. Three nodes must agree on a single value in an asynchronous network. Why is a policy of "the first proposal to arrive wins" unsafe?
2. What fundamental guarantee does a majority quorum provide, and under what conditions can asymmetric read and write quorums provide the same safety guarantee?
3. What state must an acceptor persist to durable storage across crashes, and what failure occurs if this state is lost?
4. What is the precise distinction between a value being **chosen** versus being **known to be chosen**?

Revisit your answers after completing Part III. The conceptual distance between the two marks the core contribution of this book.

PART I

One decision

We begin with one blank line in a ledger and three librarians who cannot meet. We end with three rules that decide whether any Paxos message is legal.

1 Foundations of Consensus

Learning Objectives
After completing this chapter, you will be able to:
- Distinguish safety invariants from liveness conditions in asynchronous systems.
- Calculate valid read and write quorum configurations in `membership_init` and explain why invalid sizes are rejected.
- Order ballots structured as `(round, priority, node)` triples using lexicographical integer comparison.
- State Lamport's ballot invariants B1, B2, and B3, and identify the Odin procedures that enforce each rule.
- Explain why quorum intersection guarantees safety across crashes only when acceptors persist promises and votes to non-volatile storage.

Checkpoint: Prerequisites
This chapter assumes familiarity with basic set theory, integer arithmetic, and the asynchronous network model (where messages may be delayed, duplicated, or dropped, and nodes may fail by stopping). If you already know Paxos, review the self-test exercises at the end of the chapter to verify that your mental model relies on highest-ballot selection rather than counting votes.

1.1 The empty ledger

Three librarians sit in three separate rooms. Each keeps a copy of the same ledger, and the next line of every copy is blank. Two merchants arrive at the front desk at the same moment. One wants the line to read `olive_oil = 50`; the other wants `olive_oil = 80`. Each merchant hires runners to carry slips to the librarians.

The librarians cannot leave their rooms; they talk only through runners, and the runners are unreliable. A runner may take an hour or a week, deliver slips out of the order they were written, deliver the same slip twice, or never arrive. The one thing a runner never does is change the words on a slip.

What we want is easy to state. At most one value may ever be marked final on that line. Tentative votes may differ; final decisions must agree. A merchant may be told a value only after it is settled. And if runners deliver, a majority of librarians stay at their desks, and one proposer can finish without repeated interruption, the line should eventually be filled. The first two wishes say what must never happen; the third says what should eventually happen. They are different kinds of promise.

Definition: Safety
Nothing bad ever happens. For one ledger line, two different values are never chosen, and a chosen value stays chosen. A safety property can be violated only by something that has already happened, so it never depends on how fast a runner is.

Definition: Liveness
Something good eventually happens. For one ledger line, some proposed value is eventually chosen, provided enough librarians are awake, enough runners deliver, and one merchant is left alone long enough to finish.

A library where nobody writes anything is perfectly safe. That sounds like a joke, but it lets us design the rules that keep us safe first, with no assumption about time, and add the rules that make progress afterwards. Every rule in this chapter is a safety rule.

Prediction Exercise

> Librarians 1 and 2 have each written `olive_oil = 50` in ink and told nobody. Librarian 3 has an empty line. Is the value chosen? Write one sentence before you read on, and do not use the phrase "the merchant knows" in it.

1.2 Three tempting answers

Each of these designs is the first thing a careful engineer proposes. Each fails, and each contributes a piece that survives into the final protocol.

First writer wins. Each librarian writes whichever slip reaches her first. Runner A reaches librarian 1 first, runner B reaches librarian 2 first, and now copy 1 says 50 and copy 2 says 80, forever. What survives: every librarian keeps local state and answers from it. What fails: nothing ties the local decisions together.

One master. Appoint librarian 1 as the only writer; the others copy her. This works until she falls asleep. If librarian 2 takes over, she must know whether librarian 1 already wrote something that reached a copy, and she cannot ask a sleeping colleague. What survives: one active proposer at a time keeps the protocol simple. What fails: a takeover has no safe way to learn the past.

Unanimity. Write a value only when all three librarians agree. Nobody can ever disagree, but if one librarian is asleep or one runner is lost, the line stays blank forever. What survives: a value is settled by a **set** of acceptances, not by one person. What fails: the set is too large to survive a single absence.

The final protocol combines the three survivors: durable local state, ordered attempts by proposers, and quorums that connect each new attempt to earlier decisions. Concurrent proposers can delay progress, but must not break agreement.

> **Exercise 1.1**
> A cluster has four voters. Write down two sets of two voters that do not intersect. Describe one execution in which each set accepts a different value for slot 1, and name the invariant that fails.

1.3 The failure model

A proof is only as good as the world it assumes. This library assumes four rules.

1. **Nodes may crash and recover.** A node runs the algorithm exactly until it halts, and it may halt between any two instructions. After halting it says nothing. It comes back only if the host rebuilds it from a durable journal, and is then the same member only because it remembers what it wrote.
2. **The network is asynchronous.** There is no bound on how long a message takes. The core reads no clock; the host feeds it logical ticks, which affect only liveness.
3. **Runners lose, duplicate and reorder.** Any message may be dropped, delivered twice, or delivered after a message sent later. Every handler in `src/election.odin` and `src/consensus.odin` must be harmless under duplicates and correct under reordering.
4. **Nobody lies.** A delivered message is exactly what its sender wrote, and the sender followed the algorithm: the non-Byzantine assumption. `node_step` rejects senders outside the membership with `.Not_Member` but does not authenticate; that is the host transport's job.

> **Warning: The disk is part of the algorithm**
> Rule one hides the whole difficulty. A node that halts and forgets can break a promise it already made. The only defence is to make the promise durable before anyone else can act on it; the section on durable storage shows the code.

1.4 Quorums

We cannot wait for everyone, and we cannot let anyone act alone. A **quorum** is large enough to matter and small enough to assemble.

> **Definition: Quorum**
> A set of acceptors whose acceptance settles a value. The defining property is not size but overlap: any quorum used to read the past and any quorum used to write a value share at least one acceptor.

With N acceptors and simple majorities, a quorum has at least $\lfloor \frac{N}{2} \rfloor + 1$ members. Two majorities of an N-element set cannot be disjoint, because together they would hold more than N elements. The overlapping member is the witness who carries the past into the future.

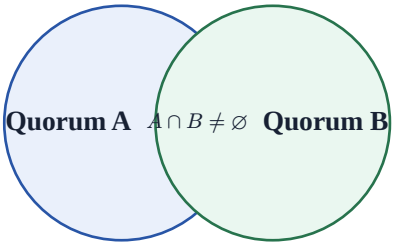


Figure 1: Any two majority quorums share at least one acceptor. That acceptor is the only link between a value chosen earlier and a leader elected later.

The library does not hard-code majorities. It stores a read quorum size for phase one and a write quorum size for phase two, defaults both to the majority, and validates only the property that matters:

src/membership.odin

```
majority := total / 2 + 1
read := read_quorum_override if read_quorum_override != 0 else majority
write := write_quorum_override if write_quorum_override != 0 else majority
if read <= 0 || read > total do return .Invalid_Read_Quorum
if write <= 0 || write > total do return .Invalid_Write_Quorum
if read + write <= total do return .Non_Intersecting_Quorums
validated.read_quorum_size, validated.write_quorum_size = read, write
```

Odin's integer division makes `5 / 2 + 1` evaluate to `3`: five members with default quorums need three promises and three acceptances. A host may lower the write quorum to two only if it raises the read quorum to four, because the sum must exceed `total`. The error names the consequence, not the arithmetic: `.Non_Intersecting_Quorums`. Why odd counts? Three acceptors need two and survive one crash. Four need three and still survive only one: the fourth member costs a machine, a journal and a link and does not increase the number of crashes tolerated by majority quorums. Five need three and survive two. Voting groups are usually three or five.

1.4.1 Intersection is not memory

Suppose acceptors 1 and 2 accept `olive_oil = 50`. Later a new leader asks acceptors 2 and 3 what they have accepted. The sets intersect at acceptor 2, so on paper the leader must learn about the 50.

Now suppose acceptor 2 kept its vote only in RAM and lost power in between. It restarts with an empty line and says it has never voted. The new leader proposes 80 to acceptors 2 and 3, they accept, and the ledger has chosen two values. The intersection still exists; the knowledge does not. Overlap is a property of sets; safety also needs a property of memory, the subject of the section on durable storage.

1.5 Ballots

Because proposers fail, we must allow many attempts to fill one line and be able to say which attempt is later. Each attempt is a **ballot**; ballots are unique and totally ordered.

src/ballot.odin

```
// A ballot is one 64-bit integer, so B1 (a total order on ballots) is integer
// comparison and every message and record carries eight bytes:
//
```

```
//   bits 63..24  round      (40 bits, the campaign counter; round 0 is reserved for
//                            slot owners under rotating ownership)
//   bits 23..16  priority   (8 bits, breaks ties between rounds)
//   bits 15..0   node       (16 bits, the proposer; makes every ballot unique)
Ballot :: distinct u64

BALLOT_ZERO      :: Ballot(0)
BALLOT_ROUND_BITS :: 40
MAX_ROUND        :: u64(1) << BALLOT_ROUND_BITS - 1

ballot_make :: #force_inline proc(round: u64, priority: u8, node: Node_Id) -> Ballot {
    return Ballot(round << 24 | u64(priority) << 16 | u64(node))
}

ballot_round :: #force_inline proc(b: Ballot) -> u64 {
    return u64(b) >> 24
}

ballot_priority :: #force_inline proc(b: Ballot) -> u8 {
    return u8(u64(b) >> 16)
}

ballot_node :: #force_inline proc(b: Ballot) -> Node_Id {
    return Node_Id(u64(b))
}
```

A ballot is a triple `(round, priority, node)` packed into one unsigned integer, with the round in the high 40 bits, the priority in the next 8 and the 16-bit `Node_Id` in the low bits. Because each field sits above the fields that rank below it, the plain integer order `<` on two ballots is exactly the lexicographic order on the triples. A greater `round` always wins. Within a round, a greater `priority` wins; the host sets it through `Node_Options.priority` to prefer some members as leaders. Within a round and a priority, the greater `node` id wins, and because node ids are unique inside a membership, two members never produce the same ballot. `ballot_make` packs the triple and `ballot_round`, `ballot_priority` and `ballot_node` unpack it; everything else in the library compares ballots with `<` and `==`. Forty round bits are enough for `MAX_ROUND` campaigns before `.Ballot_Exhausted`; `BALLOT_ZERO` is `Ballot(0)`, below every ballot a campaign can produce. Round 0 itself is reserved for slot owners under rotating ownership, which a later chapter covers.

Left	Right	`left < right`
`(4, 0, 2)`	`(5, 0, 1)`	`true`: the higher round dominates.
`(7, 2, 9)`	`(7, 3, 1)`	`true`: same round, priority decides.
`(9, 0, 1)`	`(9, 0, 4)`	`true`: same round and priority, node id decides.

Exercise 2.1
Order the ballots (round 2, priority 0, node 1), (1, 5, 3), (2, 0, 3), (1, 5, 1). Which one wins a contest, and why does priority sit between round and node?

1.6 Votes and the meaning of "chosen"

An acceptor **votes** by accepting a proposal: it records a ballot and a value for the slot, in the `vote_ballot` and `value` columns of its `Ledger`, and holds at most one vote per slot, the latest.

Definition: Chosen

> A value v is chosen for a slot when a write quorum of acceptors has each accepted v under the same ballot. Being chosen is a fact about the acceptors' durable state. It does not require any proposer, learner or client to know that it happened.

The moment the last acceptor of a write quorum makes its vote durable, the value is chosen, even if that acceptor's reply is lost, even if the leader dies in the next microsecond, even if no learner ever hears. Every later rule exists to make sure a fact nobody knows about is still respected.

> **Prediction Exercise**
> A leader collects acceptances for `olive_oil = 50` from acceptors 1 and 2 out of three, then crashes before it sends a single commit. A new leader starts a ballot. What value must the new leader end up proposing, and which acceptor will tell it?

1.7 Three invariants

Lamport's proof of the Synod protocol rests on three conditions on ballots, each of which the library keeps in a specific place.

Rule	Statement	Where the library keeps it
B1	Every ballot is unique.	`start_campaign` builds `ballot_make(greatest + 1, node.priority, node.id)`; `membership_init` rejects `.Duplicate_Node_Id` and `.Invalid_Node_Id`.
B2	Every phase-one quorum intersects every phase-two quorum.	`membership_init` returns `.Non_Intersecting_Quorums` unless `read_quorum_size + write_quorum_size > total`.
B3	If any acceptor in the phase-one quorum has voted, the new ballot proposes the value of the greatest-ballot vote reported.	`on_promise` keeps only the greatest vote per slot, and a reported decision dominates every vote; `resolve_chunk` passes that value to `send_accept`.

B1 makes "later" well defined. B2 makes sure a later ballot cannot avoid meeting a witness. B3 tells the later ballot what to do with what the witness says, and it is the rule people get wrong. It is not "the value with the most votes" and not "the most recent value you heard". Among the votes reported by your read quorum, find the one with the greatest ballot and propose its value; if nobody reported a vote, propose what you like. Older votes may belong to attempts that never reached a quorum, so counting them counts noise. The proof below shows that the greatest-ballot vote carries the chosen value whenever a choice was made.

1.8 The greatest-vote proof

Here is the proof that B1, B2 and B3 together keep safety. The single-decree chapter uses it to explain why each message field exists, and the safety-argument chapter restates it as formal lemmas, each mapped to the procedure that keeps it.

Claim. Suppose value v is chosen at ballot b, so a write quorum W of acceptors each accepted (b, v). Then every ballot $b' > b$ whose leader sends an Accept sends the value v.

Proof. By strong induction on b'. Fix $b' > b$ and assume the claim for every ballot strictly between b and b'.

The leader of b' finished phase one, so a read quorum R promised b'. By B2, R and W share some acceptor a. Acceptor a did two things: it accepted (b, v) and it promised b'. Had it promised b' first, then when (b, v) arrived, $b < b'$ would have been below its promise and it would have refused, contradicting $a \in W$. So a accepted (b, v) **before** promising b'.

When a answered the prepare for b', its slot therefore held a vote with ballot at least b: either (b, v) itself, or a later vote at some b'' with $b < b'' < b'$ that overwrote it. By the induction hypothesis every such b'' carried v. So the greatest-ballot vote reported by R has ballot at least b and value v, and by B3 the leader of b' proposes v. ■
Three facts had to hold, and each is a line of code. Acceptors refuse ballots below their promise: `on_accept` calls `send_nack` when `msg.ballot < l.promised`, where `l` is the acceptor's `Ledger`. Acceptors report their vote in the promise: `on_prepare` sends one `Promise_Message` per used cell. And acceptor a still remembered both when asked, which is where durable storage enters.

Exercise 4.2
Complete the reasoning for ballot succession: Suppose value x was chosen at ballot 12 by write quorum W. A subsequent leader at ballot 20 collects promises from read quorum R.

1. Because R and W intersect at acceptor a, and a could not have promised ballot 20 before accepting ballot 12 (otherwise it would have rejected the proposal at ballot 12), what is the minimum ballot number a will report in its promise?
2. By the inductive hypothesis, what value must any reported vote with a ballot strictly between 12 and 20 carry?
3. What value must the greatest-ballot reported vote carry?
4. Which ballot invariant (B1, B2, or B3) obligates the leader at ballot 20 to propose this value?

Hint: Review the induction step in the greatest-vote proof above.

1.9 Why durable storage is mandatory

The proof used the phrase "still remembered". An acceptor that votes, replies, and then loses the vote to a power failure has told the leader something no longer true. Worse, an acceptor that promises b', replies, and then forgets may later accept a ballot below b', which is exactly the refusal the proof relied on.
Lamport's priests wrote in indelible ink. Here, a promise or a vote is a `Write` record the host must append to a journal and sync before any message from the same transition leaves the machine. The `Ledger` that holds the durable state refuses to move backwards:

src/ledger.odin

```
ledger_apply :: proc(l: ^Ledger($Value, $WINDOW), write: Write(Value)) -> Error {
    switch w in write {
    case Write_Promise:
        if w.ballot < l.promised do return .Promise_Regression
        l.promised = w.ballot
    // ...
    case Write_Vote(Value):
        if w.slot == 0 do return .Invalid_Slot
        if w.ballot < l.promised do return .Promise_Regression
        // ...
```

`.Promise_Regression` is not a protocol message. It is a self-check: a `Write_Promise` or `Write_Vote` that would lower the promised ballot is refused, because the protocol never produces one. A host that sees it has a journal written out of order or corrupted, and the hint in `explain_error` says to stop the node. The other half of indelible ink lives in `Effects`: every transition returns its writes and messages in one batch, and with the default `Durability_Gate.Enforced` the batch refuses to hand out messages until the host calls `confirm_writes_durable`. The single-decree chapter walks through that gate at every crash point.

Exercise 4.1
A node writes `Write_Promise` for ballot (5, 0, 2) but crashes before the write is synced, then restarts and receives Prepare for ballot (4, 0, 3). What may it answer, and which rule of the host contract decides?

1.10 From rules to messages

We can now derive the messages from their purpose. Read the invariants as obligations on a proposer and ask what evidence it must request and send.

1. B1 says: pick a ballot greater than any you have seen. That needs no message, only a memory of the greatest round observed.
2. B3 needs the votes of a read quorum, and the proof needs those acceptors to refuse everything below your ballot from now on. One message asks for both: **Prepare**, carrying the ballot. The reply, **Promise**, carries the acceptor's vote for the slot if it has one.
3. B2 says: count promises until you have a read quorum, then apply B3 and pick.
4. Send the ballot and the value to the acceptors: **Accept**. An acceptor that has not promised anything greater records the vote durably and replies **Accepted**.
5. Count Accepted replies until you have a write quorum. The value is chosen. Tell everyone: **Commit**.

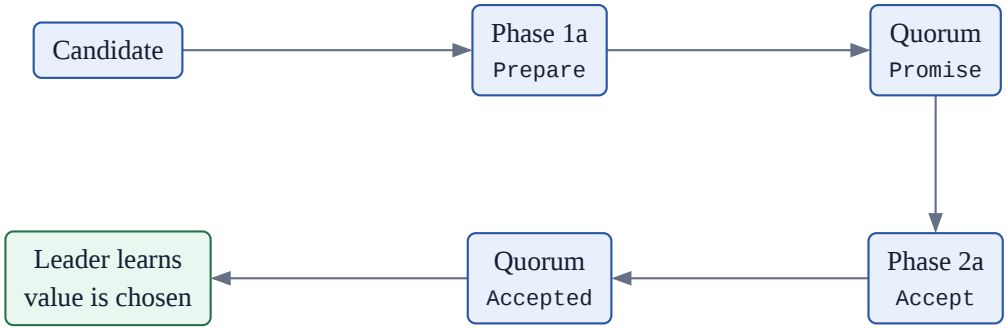


Figure 2: The five messages in order. Phase one earns the right to propose and learns the past; phase two writes the future.

One more message follows from liveness rather than safety. An acceptor that receives a Prepare or an Accept below its promise answers **Nack** with the ballot it has promised. This lets the proposer react promptly instead of waiting for a timeout. The single-decree chapter shows the exchange.

Checkpoint: Synthesis Check

Before proceeding to the single-decree protocol, verify your understanding of these core questions:

1. Why does a four-member cluster tolerate no more crash failures than a three-member cluster under majority quorums?
2. Which ballot invariant (B1, B2, or B3) does the error `.Non_Intersecting_Quorums` enforce?
3. If a read quorum reports three votes: `((3, 0, 1), apple)`, `((9, 0, 2), apple)`, and `((7, 0, 3), pear)` - which value must the new leader propose, and why?
4. What safety violation occurs if the acceptor that reported `((9, 0, 2), apple)` held that vote in volatile RAM and rebooted before responding?

Review & Discussion

Explain why a simple majority agreement is insufficient to preserve consistency across crashes and leader transitions:

- How an acceptor crash can erase volatile state unless writes are committed to durable storage.
- How overlapping read and write quorums guarantee that at least one surviving acceptor witnessed prior votes.
- Why the new leader must adopt the value with the **highest ballot number** rather than the most frequently reported value.
- Why `Write_Promise` and `Write_Vote` must be synced to disk before acknowledging transitions.

PART II

The complete ballot

We run one ballot from the first Prepare to the last Commit. We stop at every write, every message and every place the power can fail, and we show that the restart never has to guess.

2 The Single-Decree Protocol

Learning Objectives
After completing this chapter, you will be able to:
- Trace a complete single-decree ballot through the library's protocol event handlers.
- Precisely distinguish between the four states of a value: **accepted**, **chosen**, **committed**, and **applied**.
- Order disk writes and network transmissions according to the persist-before-send contract.
- Explain how Nack responses trigger fast campaign termination without waiting for election timeouts.
- Reconstruct consistent node state from the replay of durable `Write` records after a crash at any execution point.

Checkpoint: Foundational Invariant
Recall ballot invariant B3: A proposal at ballot b must adopt the value of the highest-ballot vote reported by a phase-one read quorum, or may propose an arbitrary fresh value only if no votes were reported. Every phase-two proposal rule in this chapter derives directly from this invariant.

2.1 The four roles

Paxos is described with four roles. Keep them apart in your head even though this library runs all four inside one `Node` value.

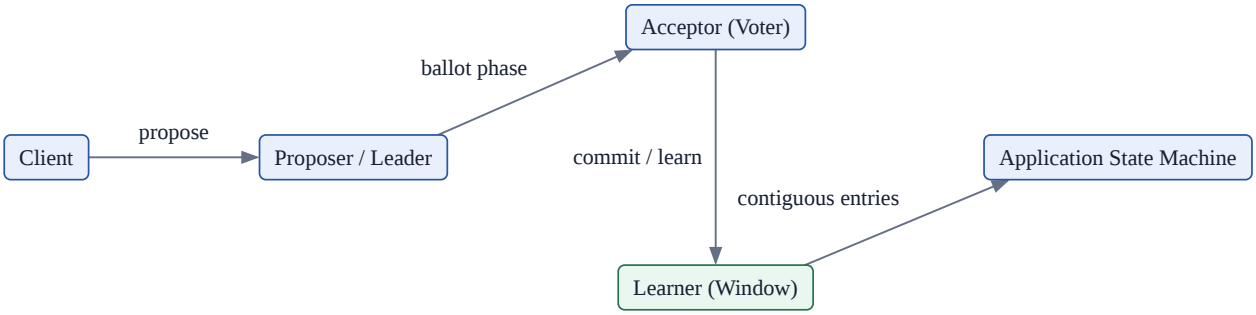


Figure 3: A client supplies intent. The proposer orders it. The acceptors are the durable memory. The learner releases decided entries, in order, to the application.

The **client** wants a command applied; it is outside the library, and the host calls `paxos.propose` on its behalf. The **proposer** runs campaigns, picks values under B3, and drives phase two; its bookkeeping is volatile, and a restarted proposer simply campaigns again higher. The **acceptor** promises and votes, and never says anything it has not first written down. The **learner** collects commits and hands the application a contiguous prefix of decided slots, never a slot with a hole below it.

One `Node(Value, MAX_MEMBERS, WINDOW_SLOTS, CHUNK_SLOTS, GATE)` plays all four. Its proposer side is the `Role` enum: `.Follower` acts only as acceptor and learner, `.Preparing` is running phase one, `.Leader` may run phase two. A node built with `paxos.init_learner` is the exception: its id lies outside the membership, it never promises or votes, and it answers `.Learner_Message_Forbidden` to anything but Commit.

2.2 Phase one: earn the right to propose

Phase one has two purposes, both from the foundations chapter. It secures a read quorum's promise to refuse lower ballots, and it collects that quorum's votes so that B3 can be applied.

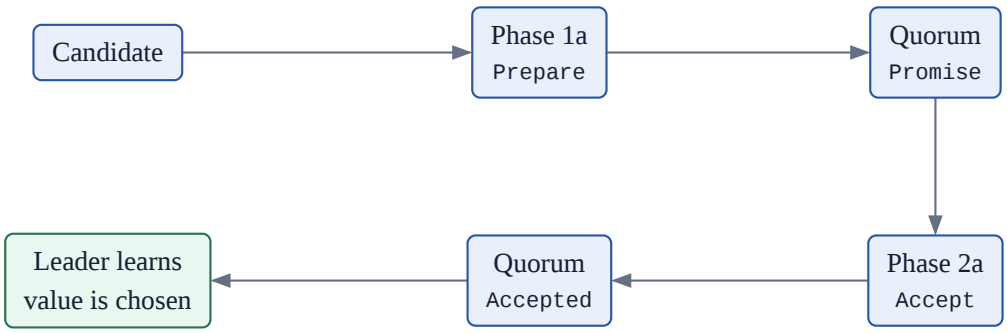


Figure 4: Phase one asks the past; phase two writes the future. Each arrow is one variant of `Message(Value)`.

2.2.1 Prepare

A campaign starts when the host calls `paxos.campaign(&node, noop, &effects)` or when a follower's election timer expires inside `paxos.tick`. Both paths reach `start_campaign`:

src/election.odin

```
    greatest := max(node.highest_observed_round, ballot_round(node.ballot))
    greatest = max(greatest, ballot_round(ledger_highest_ballot(&node.ledger)))
    if greatest >= MAX_ROUND do return .Ballot_Exhausted

    node.ballot = ballot_make(greatest + 1, node.priority, node.id)
    node.role = .Preparing
```

The new round is one more than the greatest round this node has seen from any source: its own last ballot, any round a peer has mentioned, and the greatest ballot anywhere in its own ledger, which `ledger_highest_ballot` takes over the promise, every per-slot promise and every vote. That is B1 in one expression. The candidate then promises itself, setting `ledger.promised` to the new ballot and adding a `Write_Promise` to the batch, and calls `broadcast_all` with a `Prepare_Message` holding `ballot`, the slot range `first..last` it wants to hear about, and a `scope` of `.Global`, meaning "promise me every slot from `first` on"; it proposes no value. The order matters: the promise is in the same batch as the `Prepare`, so it is durable before any peer hears the ballot, and a candidate that crashes can never build the same ballot again. The broadcast includes the node itself, so the candidate's own acceptor answers through the same handler as everyone else's; by then the promise is already recorded and `on_prepare` writes nothing new.

> **API anchor: `paxos.campaign(node, noop, effects)`**
> Starts phase one. Returns `.Not_Voter` for a learner and `.Campaign_Disabled` for an acceptor-only member. The `noop` value is remembered for filling recovered holes. in `node_campaign` in `src/election.odin`

2.2.2 Promise

An acceptor compares the Prepare ballot with the greatest ballot it has ever promised, which lives in `node.ledger.promised`:

src/election.odin

```
    l := &node.ledger
    if msg.ballot < l.promised {
        send_nack(node, from, msg.ballot, l.promised, 0, effects)
    return .None
}
    if msg.first == 0 || msg.last < msg.first do return .Invalid_Slot
    switch msg.scope {
    case .Global:
        if msg.ballot != l.promised {
```

```
            l.promised = msg.ballot
            effects_add_write(effects, Write_Promise{msg.ballot})
    }
        observe_leader(node, from, msg.ballot)
```

A ballot below the promise is refused with a Nack. A ballot equal to the promise is a retransmission and needs no new record. A greater ballot becomes the new promise, and `Write_Promise` enters the effects batch before any reply. (The `.Bounded` scope, which promises one slot range instead of every slot, belongs to rotating ownership and waits for a later chapter.) The handler then walks the ledger's `used` bitmap and reports every vote or decision it holds in the requested range, one `Promise_Message` per cell:

src/election.odin

```
    cell, used := bit_set_next(l.used, 0)
    for used {
        slot := l.slot[cell]
        if slot > l.anchor.chosen_trim_slot && slot >= msg.first {
            if slot > msg.last {
                more = true
            } else {
                    reported += 1
                    send_to(node, from, effects, Promise_Message(V){
                        ballot = msg.ballot, slot = slot, vote = l.vote_ballot[cell],
                        state = l.state[cell], value = &l.value[cell],
                    })
            }
    }
        cell, used = bit_set_next(l.used, cell + 1)
}
```

The `state` field is the cell's `Cell_State`. A cell this node voted in is reported as `.Voted` with its `vote` ballot. A cell this node learned through a Commit is reported as `.Chosen`: it carries a value and no meaningful ballot, and it tells the candidate "this is decided; do not run phase two for it". After the per-slot messages comes one `Promise_Range_Message`, which names the slot range answered, the `reported` count of per-slot promises sent, and the acceptor's own decided prefix. The candidate counts an acceptor toward the read quorum only when it holds the range descriptor **and** that many per-slot promises, so reordered runners cannot make a partial answer look complete.

> **Write before speak**
> `Write_Promise` is added to `effects.writes` before `Promise_Message` is added to `effects.messages`, and the host may not read the messages until it has confirmed the writes. This is the point where the acceptor's word becomes indelible.

2.2.3 Value selection

The candidate's `on_promise` stores each report in the `recovered_ballot`, `recovered_state` and `recovered_value` columns at the chunk-relative index `slot - recover_base`. This index is distinct from a ledger cell. It replaces a vote only when `msg.vote > node.recovered_ballot[cell]`, so each slot keeps the greatest ballot seen; a report with `state = .Chosen` dominates every vote. Two reports with equal ballots and different values return `.Conflicting_Value`: that would mean B1 was broken, and the library stops rather than pick. When `maybe_resolve_chunk` counts a read quorum of complete answers, `resolve_chunk` walks the slots and applies B3 to each:

src/election.odin

```
        cell, in_chunk := recovery_index(node, slot)
        assert(in_chunk, "Recovery slot outside chunk. Hint: Report this invariant
failure.")
        if chosen, is_chosen := ledger_chosen_at(&node.ledger, slot); is_chosen {
            broadcast_peers(node, effects, Commit_Message(V){slot = slot, value = chosen})
        } else if node.recovered_slot[cell] == slot && node.recovered_state[cell]
== .Chosen {
            record_commit(node, slot, node.recovered_value[cell], effects) or_return
            if decided, ok := ledger_chosen_at(&node.ledger, slot); ok {
                broadcast_peers(node, effects, Commit_Message(V){slot = slot, value
= decided})
            }
        } else {
            value := node.noop.?
            if node.recovered_slot[cell] == slot && node.recovered_state[cell] == .Voted {
                value = node.recovered_value[cell]
            }
            accept_err := send_accept(node, slot, node.ballot, value, effects)
            if accept_err == .Not_Leader {
                // A higher ballot already holds this decree: this candidate lost.
Step down
                // quietly; the winner (or the next timeout) finishes the range.
                node.role = .Follower
                return false, .None
            }
            accept_err or_return
        }
```

A slot the candidate itself already holds as chosen is re-announced. A candidate whose accept is refused because a higher ballot already promised that decree has lost the election; it steps down rather than report an error. A reported decision is committed outright. A reported vote is re-proposed under the new ballot. A slot nobody voted in, below one somebody did, is filled with the host's no-op. For a fresh single decision none of these fire: `become_leader` runs, the role becomes `.Leader`, and `next_slot` is 1.

> **Prediction Exercise**
> The candidate receives an acceptor's `Promise_Range_Message` with `reported = 1` before the matching `Promise_Message` arrives. May it count that acceptor toward the read quorum yet? Name the vote it might miss and the rule that vote protects.

2.3 Phase two: write the future

A client value arrives through `paxos.propose(&node, value, &effects)`, which checks `node.role == .Leader`, takes `next_slot`, and calls `send_accept(node, slot, node.ballot, value, effects)`.

2.3.1 Accept

src/consensus.odin

```
    l.promised_at[cell] = max(l.promised_at[cell], ballot)
    ledger_record_vote(l, cell, ballot, value)
    effects_add_write(effects, Write_Vote(V){ballot = ballot, slot = slot, value =
&l.value[cell]})

    if bit_set_insert(&node.acknowledgements[cell], node.self_index) do
node.acknowledged[cell] += 1
    if membership_write_quorum(&node.membership) == 1 {
        record_commit(node, slot, value, effects) or_return
```

```
    }
    broadcast_peers(node, effects, Accept_Message(V){ballot = ballot, slot = slot, value
= &l.value[cell]})
```

The leader is also an acceptor, and it votes for its own proposal without sending itself a message: `ledger_record_vote` puts the ballot and value into its own ledger, `Write_Vote` records that vote, and the leader sets its own bit in the cell's `acknowledgements` and counts it in `acknowledged`. The proposal **is** that vote: `Accept_Message.value` points into the leader's ledger, and the pointer stays valid until the node's next transition, which is why a host serialises the batch before running another. This is the **local acceptance optimisation**. With three members and a write quorum of two, one remote Accepted completes the quorum. `broadcast_peers` sends the Accept to everyone except the leader.

> **API anchor: `paxos.propose(node, value, effects)`**
> Returns the slot the value took, or `.Not_Leader` if phase one has not completed, `.Window_Full` if the bounded window has no room, and `.Leader_Catching_Up` if the host asked proposals to wait for the inherited prefix. in `node_propose` in `src/consensus.odin`

2.3.2 Accepted

`on_accept` checks the promise once more, because another candidate may have campaigned since the leader's phase one; a ballot below the effective promise for the cell, `l.promised` or the cell's own `promised_at`, whichever is greater, gets `send_nack` naming the slot. Otherwise the acceptor raises the cell's `promised_at` to the accepted ballot (a vote implies the promise, even if no Prepare was ever seen), stores the vote with `ledger_record_vote`, adds `Write_Vote` to the batch, and only then sends `Accepted_Message` with the ballot, the slot, and `decided_through`, its own contiguous decided prefix. The leader records that prefix per peer and uses it in the Multi-Paxos chapter to decide what to retransmit. A duplicate Accept for a vote already held under the same ballot is answered with Accepted again and no new write. A cell that is already `.Chosen` never votes again: an Accept for the same value gets Accepted, an Accept for a different value gets the decision back as a `Commit_Message`. One more guard applies before any of this: an Accept whose ballot has round zero is accepted only from the slot's owner under rotating ownership, and otherwise ignored; a later chapter explains that mode.

2.3.3 Commit

The leader's `on_accepted` ignores replies for any slot and ballot but the one it is driving in that cell (`lead_slot` and `lead_ballot`), adds the sender's bit to `acknowledgements[cell]`, and returns until `acknowledged[cell]` reaches `membership_write_quorum`. At that count the value is chosen. `record_commit` marks the cell `.Chosen` with `ledger_record_chosen`, adds `Write_Chosen`, and calls `emit_contiguous`, which walks from `delivered_through + 1` upward and appends every chosen slot it finds to `effects.committed` until it reaches a hole. Then `broadcast_peers` sends `Commit_Message{slot, value}` to every peer, whose `on_commit` runs the same `record_commit` and the same `emit_contiguous`. A later Accepted for a slot already chosen returns early.

> **Four words that are not synonyms**
> **Accepted**: one acceptor holds a durable vote. **Chosen**: a write quorum holds durable votes under one ballot; nobody needs to know. **Committed**: a node holds a durable `Write_Chosen` for the slot. **Applied**: the host has consumed the entry from `committed_slice` and changed its state machine. Safety is about chosen; the rest is delivery.

> **API anchor: `paxos.step(node, envelope, effects)`**
> Processes one message addressed to this node: `.Wrong_Recipient` if `envelope.to` is not this node, `.Not_Member` if `envelope.from` is outside the membership, otherwise a dispatch on the `Message(Value)` variant. in `node_step` in `src/consensus.odin`

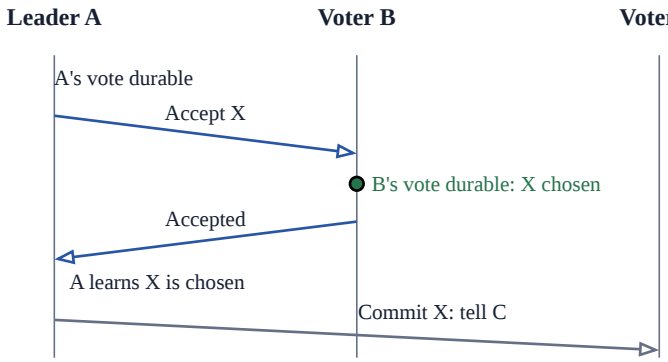


Figure 5: Three voters, write quorum two. X becomes chosen when B makes the second vote durable. The leader learns this later. A delayed acknowledgement changes knowledge, not the chosen value; C need not have voted yet. Time runs downward.

2.4 A complete trace

Members 1, 2 and 3; read and write quorums both 2; every priority 0. The host calls `paxos.campaign(&n1, noop, &effects)`, later `paxos.propose(&n1, tea, &effects)`. The runner to member 3 is slow. Each row lists writes before messages, the host's order, and drops the `_Message` suffix from message names. Ballots are written as their unpacked `(round, priority, node)` triples, and the chunk is the default 64 slots.

Step	Actor	Event and reason
1	N1	`start_campaign`: ballot `(1, 0, 1)`, role `.Preparing`, `recover_base = 1`, `recover_last = 64`. Writes `Write_Promise{(1,0,1)}`. After the host makes it durable, sends `Prepare{(1,0,1), first = 1, last = 64, scope = .Global}` to 1, 2, 3.
2	N1	`on_prepare` on its own Prepare: `(1,0,1)` equals the promise already written by the campaign. No new write and no used cells, so no per-slot promise. Sends `Promise_Range{reported = 0}` to 1.
3	N2	`on_prepare`: writes `Write_Promise{(1,0,1)}`, `leader_hint = 1`. Sends `Promise_Range{reported = 0}` to 1.
4	N1	`on_promise_range` from itself: one of two complete. No effects.
5	N1	`on_promise_range` from N2: read quorum met. `resolve_chunk` finds no votes. `become_leader`: role `.Leader`, `next_slot = 1`. No effects.
6	N1	`node_propose(tea)`: slot 1. `send_accept` records its own vote, writes `Write_Vote{(1,0,1), 1, tea}` and marks its own acknowledgement (`acknowledged = 1` of 2). Sends `Accept{(1,0,1), 1, tea}` to 2 and 3.
7	Host of N1	Appends the record, syncs, calls `confirm_writes_durable`, then reads `messages_slice` and sends. Reading first would stop the process.
8	N2	`on_accept`: `(1,0,1)` is not below `promised`. Writes `Write_Vote{(1,0,1), 1, tea}`. Sends `Accepted{(1,0,1), 1, decided_through = 0}` to 1.
9	N1	`on_accepted` from N2: acknowledgements `{1, 2}`, `acknowledged = 2`, write quorum met. **Tea is chosen.** `record_commit` writes `Write_Chosen{1, tea}`; `emit_contiguous` releases `Committed{1, tea}`, `delivered_through = 1`. Sends `Commit{1, tea}` to 2 and 3.
10	Host of N1	Persists the decision (a cheaper barrier is allowed: no promise or vote is in this batch), sends the Commits, applies tea.
11	N2	`on_commit`: `record_commit` writes `Write_Chosen{1, tea}` and releases `Committed{1, tea}`. N2 applies tea.

Step	Actor	Event and reason
12	N3	The slow Prepare arrives. Writes `Write_Promise{(1,0,1)}`, sends `Promise_Range` to 1, which ignores it: N1 is no longer `.Preparing`.
13	N3	Accept arrives: writes `Write_Vote`, sends `Accepted`. N1 records the third acknowledgement and returns early: slot 1 is already chosen.
14	N3	Commit arrives: `Write_Chosen{1, tea}`, `Committed{1, tea}`. All agree.

Look at step 9. Tea was chosen the instant N2's `Write_Vote` in step 8 became durable, because at that moment two of three acceptors held durable votes under `(1, 0, 1)`. Step 9 is N1 **learning** that fact. Had N1 crashed between steps 8 and 9, tea would still be chosen, and any future read quorum would meet the choosing quorum `{N1, N2}`. At least one witness would preserve tea through the highest-vote rule.

> **Prediction Exercise**
> Swap steps 12 and 13, so N3 receives Accept before it has ever seen a Prepare. Which branch of `on_accept` runs, what does N3 write, and what does `promised` hold afterwards?

2.5 Rejection and competing campaigns

A ballot below an acceptor's promise is answered with a Nack rather than silence, so that a stale leader learns it is stale. `Nack_Message` carries the `rejected` ballot, the `promised` ballot that beat it, the `slot` whose Accept was refused (zero for a refused Prepare), and the acceptor's `decided_through`. The receiver's `on_nack` is short because a Nack changes nothing durable:

src/consensus.odin

```
    node.highest_observed_round = max(node.highest_observed_round,
ballot_round(msg.promised))
    if msg.rejected != node.ballot || msg.promised <= node.ballot do return
    node.role = .Follower
    node.leader_hint = ballot_node(msg.promised)
```

Every Nack raises `highest_observed_round`, so the next `start_campaign` jumps above the rival instead of colliding with it again. A Nack for some other ballot, or whose `promised` is not actually greater, changes nothing else. A genuine Nack demotes the node to `.Follower` and points `leader_hint` at the winner, the node id unpacked from the promised ballot.

Suppose N2 campaigns at `(2, 0, 2)` while N1 leads at `(1, 0, 1)`. N1's own acceptor sees the greater ballot, writes `Write_Promise{(2, 0, 2)}`, and `observe_leader` demotes N1 to `.Follower`. A later `paxos.propose` on N1 returns `.Not_Leader`, and any Accept of N1's still in flight is nacked by every acceptor that promised `(2, 0, 2)`.

2.6 Liveness and the dueling leaders

Two candidates can chase each other upward forever: N1 prepares round 1, N2 prepares round 2 and N1's Accepts are nacked, N1 prepares round 3 and N2's are nacked, and so on. Nothing unsafe happens, and nothing useful either. No safety rule can remove this.

The core reads no clock, so it can be simulated and replayed deterministically. Time enters only through `paxos.tick(&node, noop, &effects)`, called at a cadence the host chooses. A follower that reaches `election_timeout_ticks` without leader contact campaigns; a leader sends `Heartbeat_Messages` every `heartbeat_interval_ticks` and retransmits every `resend_interval_ticks`. All three are fields of `Node_Options`, and so are the two tools against duels. `priority` breaks ties inside a round: if N1 and N2 both reach round 5, the greater priority wins and the other steps aside on its Nack; priority never overrides a greater round, so it cannot affect safety. `campaign_disabled` removes a member from the contest: it promises and votes like any acceptor, but `paxos.campaign` returns `.Campaign_Disabled` and its timer never fires, until the host calls `paxos.set_campaign_enabled`.

> **API anchor: `paxos.tick(node, noop, effects)`**
> Advances the election, heartbeat and resend timers by one logical tick. Followers may campaign; leaders may emit heartbeats and retransmissions. The library does not randomise timeouts, read wall-clock time, or hold leases; a host builds those outside the core, and none of them may be used as a safety argument. in `node_tick` in `src/consensus.odin`

2.7 Crash points and recovery

Every transition returns its writes and messages together in one `Effects`, and the host must persist the writes before sending the messages. The library does not trust the host to remember:

src/effects.odin

```
    when G == .Enforced {
        if e.writes_pending do host_order_violation("messages_slice before
confirm_writes_durable")
}
    return small_array.slice(&e.messages)
```

That is the body of `effects_messages_slice`, where `G` is the node's `GATE` parameter. With the default gate, reading messages while a write is unconfirmed calls `host_order_violation`, which prints a diagnostic and stops the process; so does `effects_reset` on a batch with unconfirmed writes. The check is compiled out only for a `Node` declared with `Durability_Gate.Host_Managed`, whose comment lists the four obligations such a host takes on.

> **API anchor: `paxos.confirm_writes_durable(effects)`**
> Called after every record in `writes_slice(effects)` is appended and synced. It clears `writes_pending`; until then `messages_slice` is fatal. Never call it after a failed write. in `effects_confirm_writes_durable` in `src/effects.odin`

Not every batch needs the same barrier. `effects_requires_power_loss_barrier` returns true only when the batch holds a `Write_Promise`, a `Write_Promise_At` or a `Write_Vote`; decisions (`Write_Chosen`) and trim anchors are derived state a restart can rebuild, so a host may persist them more cheaply. Step 10 of the trace is such a batch. One deliberate overlap exists: `effects_pre_durable_messages` returns an iterator whose `pre_durable_next` yields only the `Accept_Message` envelopes whose ballot has a round above zero, so a host may put a leader's Accepts on the wire while its own vote is still syncing. The batch must still be confirmed before `messages_slice` is read or the next transition runs.

2.7.1 Where the power can fail

Crash between	Journal holds	After restart
Prepare received; promise not synced	old promise	As if the Prepare was never delivered. The candidate retransmits on a tick.
Promise synced; reply not sent	new promise	The acceptor answers the retransmitted Prepare from the journal and can promise nothing lower.
Accept received; vote not synced	no vote	As if the Accept was never delivered. The leader retransmits.
Vote synced; Accepted not sent	the vote	The vote counts toward "chosen" now. A later leader's phase one will see it.
Leader's Accepts sent early; its own vote not synced	no vote	Allowed only at round above zero: the restarted proposer campaigns at a fresh

Crash between	Journal holds	After restart
		ballot, so the old Accept can only be re-proposed through phase one. At round zero an owner would reuse the same ballot, so `pre_durable_next` never releases such an Accept before the barrier.
Write quorum reached; commit not recorded	votes on a quorum	The value is chosen. Any later leader is forced by B3 to re-propose it.
Commit synced; entry not applied	the commit	Restart releases the slot again through `committed_slice`; the host applies idempotently or checks its own applied index.

No row says "guess". Every restart sees either the old durable state or the new one, and both are states the protocol could have been in.

2.7.2 What a restart replays

The host feeds every journal record, in journal order, through `ledger_replay_fold` into an empty `Ledger`:

src/ledger.odin

```
ledger_replay_fold :: proc(l: ^Ledger($Value, $WINDOW), write: Write(Value)) -> Error {
    switch w in write {
    case Write_Promise:
        l.promised = max(l.promised, w.ballot)
        return .None
    // ...
```

Replay folds monotonically: a promise record raises `promised` only if greater, a vote record claims the slot's cell, raises that cell's `promised_at` to its ballot and installs the vote unless the cell already holds a decision, and decisions and trim anchors go through `ledger_apply`. The result is the greatest promise and the latest vote per slot, which is what the acceptor knew at the crash. Then `paxos.restore(&node, id, membership, ledger)` builds a `Node` around that ledger. It comes back as a `.Follower` with no leader hint and no election bookkeeping: none of that was durable and none is needed, because the next Prepare or heartbeat re-establishes it and the next campaign starts above every round the journal holds.

API anchor: `paxos.restore(node, id, membership, ledger, floor, options)`
Rebuilds a voting member from a replayed ledger. `floor` is the slot through which the host has durably consumed released entries; cells at or below it that hold only an open vote are cleared. in `node_restore` in `src/node.odin`

Exercise 8.1
Leaders at ballots (3, 0, 1) and (4, 0, 2) both send Accept for slot 1 with different values to the same three acceptors. Trace which acceptor answers Accepted and which answers Nack for every arrival order, and state which value can be chosen.

Hint: Ask first how the leader at (4, 0, 2) finished phase one without seeing a vote for (3, 0, 1).

Checkpoint: Protocol Summary
Before proceeding to Multi-Paxos, verify your grasp of the single-decree lifecycle:

1. Which durable write record must be flushed to disk before an acceptor emits a `Promise_Message`? Which before an `Accepted_Message`?
2. Why does generating a `Nack_Message` require no durable write?

3. At what exact transition does a proposed value become legally **chosen**, and at what transition does the leader or learner discover this fact?
4. What action does `ledger_replay_fold` take when encountering a promise record lower than the ledger's current promise level?

Review & Discussion

Trace the execution of a ballot from the perspective of an acceptor:

- How an incoming `Prepare_Message` is evaluated against the local promised ballot.
- The exact sequencing of writing `Write_Promise`, waiting for disk durability via `confirm_writes_durable`, and returning `Promise_Message`.
- How an incoming `Accept_Message` is validated and recorded via `Write_Vote`.
- Which responsibilities are strictly enforced by the pure consensus engine versus which are required of the host runtime.

PART III

A sequence of decisions

A service needs more than one chosen value. We arrange decisions in slots, recover an old leader's unfinished work, fill the holes it left, and release one ordered prefix to the application.

3 Multi-Paxos Log Replication

Learning Objectives
After completing this chapter, you will be able to:
- Explain how a single phase-one preparation amortizes leader election across an unbounded sequence of future log slots.
- Describe how a candidate reconciles chunked, reordered phase-one promise manifests across multiple peers.
- Identify the boundary fences that prevent a newly elected leader from overwriting established decisions.
- Fill log holes safely during recovery using host-supplied no-op entries.
- Apply sliding-window bounds and memory floor advancement to pipeline proposals without unbounded memory growth.

Checkpoint: Multi-Slot Generalization
Recall the core single-decree principle: any phase-two proposal must adopt the value associated with the highest-ballot vote reported by a phase-one read quorum (or propose a fresh command if no prior votes exist). Multi-Paxos is the continuous application of this invariant across a sequence of indexed slots.

3.1 Why Multi-Paxos?

The single-decree protocol chooses one value. A replicated log needs a chosen value in slot 1, then slot 2, then slot 3, without end. The obvious construction runs an independent single-decree instance per slot. Each instance costs a `Prepare`/`Promise` round trip with a durable promise on every acceptor, then an `Accept`/`Accepted` round trip with a durable vote on every acceptor: two round trips and two synchronous writes for every entry, before the commit is even announced.

Multi-Paxos observes that phase one does not depend on the value. A candidate can send one `Prepare` that covers every slot from some starting point onward. Once a read quorum has promised, the candidate holds a ballot that is valid for every one of those slots, and each later proposal needs only phase two: one round trip and one durable vote per acceptor. The candidate that finishes this phase one is the leader for its ballot until a higher ballot appears.

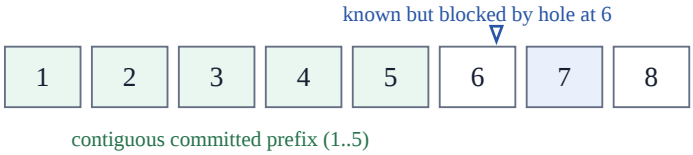


Figure 6: A log with a contiguous committed prefix (slots 1 to 5), a hole at slot 6, and a decided slot 7 that cannot be released until slot 6 is filled. Multi-Paxos runs one phase one that covers slots 6 and beyond.

3.2 One Phase One for Every Slot After `first`

Slots are 64-bit and one-based. The `Prepare` names the first slot the candidate wants resolved, the last slot of the chunk it wants reported, and a scope:

src/ballot.odin

```
// One-based position in the global decree log. Zero means "no slot".
Slot :: u64
```

src/messages.odin

```
Prepare_Scope :: enum u8 {
    Global,
```

```
    Bounded,
}

// Phase one: promise `ballot` for the decrees the scope names and report your votes in
// [first, last].
Prepare_Message :: struct {
    ballot: Ballot,
    first:  Slot,
    last:   Slot,
    scope:  Prepare_Scope,
}
```

`campaign(&node, noop, &effects)` starts the election. The candidate picks a round above every round it has observed, promised, or used, records the host's `noop` for later, sets `recover_base` to `delivered_through + 1` and `recover_last` to the end of the first chunk, and broadcasts `Prepare_Message{ballot, first = recover_base, last = recover_last}` with the default scope `.Global` to every member, itself included. The candidate answers its own `Prepare` as an ordinary acceptor when the host steps that envelope back into it; there is no private shortcut for the local vote. Everything below `first` is already delivered on this node. Everything at or above it is covered by this one promise: that is what `.Global` means, and it is the Multi-Paxos takeover this chapter describes. (`.Bounded` promises only `[first, last]` and belongs to rotating ownership, a later chapter.) The question is how an acceptor describes an unbounded suffix in a bounded message.

> **Prediction Exercise**
> A candidate wants slots 10 and above. An acceptor voted in slot 10 and in slot 12 and has nothing in slot 11. The network delivers the acceptor's reply about slot 12 first, then a summary of its reply, then the reply about slot 10. When is the candidate allowed to count this acceptor toward its read quorum? Write down your answer, then read how `Election_Peer` decides.

3.3 Combining Phase-One Replies per Slot

3.3.1 A chunk and its manifest

An acceptor answers a `Prepare` for one chunk of `CHUNK_SLOTS` slots starting at `first`. For each used cell in that chunk, it sends one `Promise_Message` carrying the slot, the cell's `state`, its `vote` ballot and its value. A cell that holds a decision (the acceptor learned it from a `Commit`, whether or not it also voted) is reported with `state = .Chosen`, so the candidate learns the decided value without a separate message kind. After the per-slot promises the acceptor sends one manifest:

src/messages.odin

```
Promise_Range_Message :: struct {
    ballot:         Ballot,
    anchor:         Trim_Anchor,
    chosen_through: Slot,
    first:          Slot,
    last:           Slot,
    reported:       u32,
    more:           bool,
}
```

`reported` says how many `Promise_Messages` the acceptor sent for the range `first..last`. `more` says whether it holds used cells above `last`, so the candidate knows another chunk is needed. `chosen_through` and `anchor` are the acceptor's two fences; we return to them below.

3.3.2 Counting a peer as complete

The candidate cannot assume the manifest arrives after the promises it describes, or before them. It keeps one `Election_Peer` per member:

src/node.odin

```
Election_Peer :: struct {
    anchor:            Trim_Anchor,
    chosen_through:    Slot,
    range_first:       Slot,
    range_last:        Slot,
    expected_in_range: u32,
    received_in_range: u32,
    range_described:   bool,
    more:              bool,
}
```

Each `Promise_Message` inside the current chunk increments that peer's `received_in_range`, but only the first time a given slot is seen from that peer: a per-peer bit set, `promise_seen`, deduplicates retransmissions. The report goes into the candidate's `recovered_slot`, `recovered_ballot`, `recovered_state` and `recovered_value` columns, which keep the highest-ballot vote per slot, with a `.Chosen` report dominating any vote; two different values under one ballot for one slot are `.Conflicting_Value`. The manifest sets `expected_in_range` and marks the peer `range_described`. `maybe_resolve_chunk` then runs after every promise and every manifest:

src/election.odin

```
    if !node.recovery_ready {
        complete := 0
        any_more := false
        for i in 0..<membership_count(&node.membership) {
            peer := &node.election[i]
            if !peer.range_described || peer.received_in_range < peer.expected_in_range
do continue
            complete += 1
            any_more ||= peer.more
        }
        if complete < membership_read_quorum(&node.membership) do return .None
        // Phase two may pause at the window boundary. Freeze the selection before any
        // vote leaves, so a late promise cannot change a value under this same ballot.
        node.recovery_ready = true
        node.recovery_more = any_more
    }
```

A peer counts only when it is fully described: its manifest arrived and every promise the manifest announced arrived too. The candidate waits until `read_quorum_size` peers are complete. That answers the prediction: the acceptor counts once its manifest and both promises are in, in whatever order they took. Wire order does not matter, and neither does a duplicate.

3.3.3 One slot, two indexes

The ledger keeps `WINDOW_SLOTS` cells; recovery scratch keeps only `CHUNK_SLOTS` reports and one chunk-sized bitmap per peer. They serve different lifetimes. The ledger survives the election; scratch describes just the chunk being recovered.

Take a window of 8 and a chunk of 3 starting at slot 7. Slots 7, 8, and 9 map to ledger cells 6, 7, and 0, but to scratch indexes 0, 1, and 2. Using the ledger mask for scratch would both exceed its bounds and confuse the next chunk with this one. `recovery_index` checks that the slot belongs to the current range before subtracting `recover_base`. The chunk need not be a power of two.

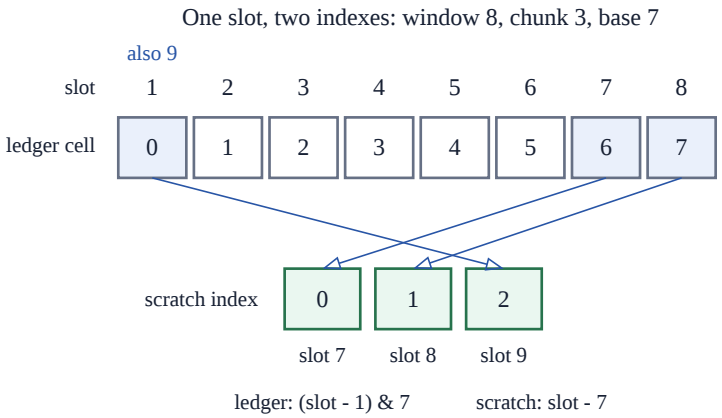


Figure 7: The arrows map each slot's ledger cell to its recovery scratch index. Slot 9 shares physical ledger cell 0 with slot 1 at different times; the slot tag and memory floor govern reuse. Recovery scratch has its own contiguous indexing.

3.3.4 Freeze the answer before acting on it

A complete read quorum gives the candidate enough evidence to select values. `recovery_ready` then freezes that selection, and `recovery_more` remembers whether another chunk is needed. This matters when the current window cannot hold every selected slot: phase two may begin, pause, and resume after the host advances the memory floor. Suppose the candidate has already sent X for one slot at ballot 5. A late promise reports an older vote for Y. Replacing X in scratch would make the retry send a second value at the same ballot. The candidate must keep its original selection. Late reports cannot change it; a later campaign selects again under a new ballot.

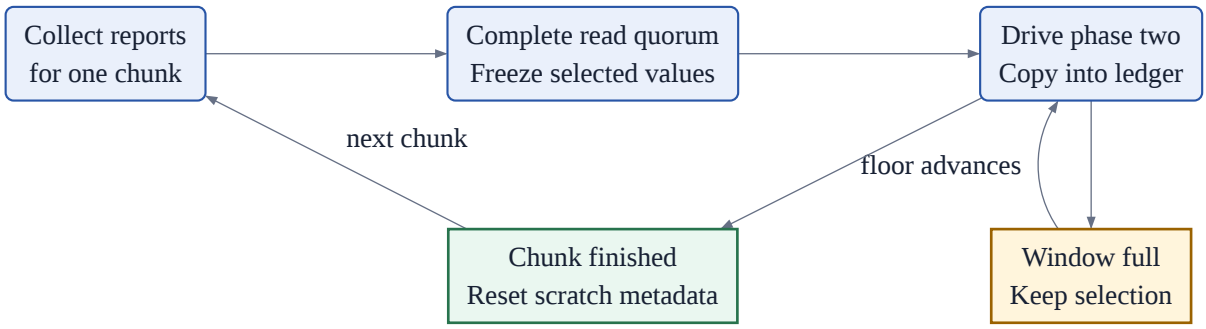


Figure 8: A full window pauses phase two without reopening value selection. Once the chunk is finished, its scratch metadata can be reset. Outgoing values point into the ledger, so clearing scratch does not invalidate the current effect batch.

> **Prediction Exercise**
> With a chunk starting at slot 7, where does slot 9 live in scratch? If phase two pauses after sending X, may a late report replace X with a higher-ballot losing vote? Explain which invariant the retry must preserve.

3.3.5 The next chunk

If any complete peer reported `more`, the candidate resolves this chunk (below) and calls `begin_next_chunk`: `recover_base` moves to the slot after the chunk and `recover_last` to the end of the next one, every `Election_Peer` is reset except its two fences, `promise_seen` is cleared, and a new `Prepare` with the new `first` and `last` goes out under the same ballot. Each reply describes at most `CHUNK_SLOTS` slots per peer. Delayed replies and retransmissions may remain in the network; the candidate only counts reports for its current ballot and chunk. When no complete peer reports `more`, the candidate becomes leader.

3.4 The Fences

Two numbers from the manifests tell the new leader where the past is already settled. `quorum_fences` folds them over the election state into a `Fences` value, starting from the candidate's own values:

src/election.odin

```
    fences.trim = node.ledger.anchor.chosen_trim_slot
    fences.chosen = node.delivered_through
    for i in 0..<membership_count(&node.membership) {
        peer := &node.election[i]
        fences.trim = max(fences.trim, peer.anchor.chosen_trim_slot)
        if peer.chosen_through > fences.chosen {
            fences.chosen = peer.chosen_through
            fences.chosen_peer = membership_get(&node.membership, i)
        }
    }
```

The trim fence is the greatest `chosen_trim_slot` any promising peer has adopted: every slot at or below it was chosen and has been released from that peer's window (the advanced-features chapter covers trim anchors). The chosen fence is the greatest `chosen_through` any promising peer reported: that peer has delivered a contiguous decided prefix through it. The leader takes the larger of the two and never re-proposes, fills, or accepts a client value at or below it. A missing vote below the fence means the slot was released, not that it is open.

The chosen fence also tells the leader that it is behind. Leadership means holding the highest ballot, not knowing every decision. When the chosen fence is above the leader's own `delivered_through`, `resolve_chunk` sends a `Learn_Message` to `chosen_peer`, asking for commits from `delivered_through + 1` for up to `CHUNK_SLOTS` slots. The leader learns those slots the same way any lagging follower does.

3.5 Holes and the No-Op

Within the chunk, above the fence, the leader must settle every slot up to the highest one it knows about, from its own cells or from recovered votes. A slot with no recovered vote is a hole: no acceptor in the read quorum voted there, so by quorum intersection nothing can have been chosen there, and the leader may propose anything. (The safety-argument chapter proves this per-slot claim as a lemma.) It must propose something, because entries are released only in contiguous order and a permanent hole would block every later slot forever.

The library does not invent a value. The host passes a `noop` to `campaign` and to `tick`; the node remembers it, and `maybe_resolve_chunk` refuses to resolve with `.Missing_Noop` if none was recorded. `resolve_chunk` then walks the chunk:

src/election.odin

```
        cell, in_chunk := recovery_index(node, slot)
        assert(in_chunk, "Recovery slot outside chunk. Hint: Report this invariant
failure.")
        if chosen, is_chosen := ledger_chosen_at(&node.ledger, slot); is_chosen {
            broadcast_peers(node, effects, Commit_Message(V){slot = slot, value = chosen})
        } else if node.recovered_slot[cell] == slot && node.recovered_state[cell]
== .Chosen {
            record_commit(node, slot, node.recovered_value[cell], effects) or_return
            if decided, ok := ledger_chosen_at(&node.ledger, slot); ok {
                broadcast_peers(node, effects, Commit_Message(V){slot = slot, value
= decided})
            }
        } else {
            value := node.noop.?
            if node.recovered_slot[cell] == slot && node.recovered_state[cell] == .Voted {
                value = node.recovered_value[cell]
            }
            accept_err := send_accept(node, slot, node.ballot, value, effects)
            if accept_err == .Not_Leader {
                // A higher ballot already holds this decree: this candidate lost.
Step down
                // quietly; the winner (or the next timeout) finishes the range.
                node.role = .Follower
```

```
                return false, .None
            }
            accept_err or_return
        }
```

Three outcomes per slot, after a slot the leader itself already holds as chosen is simply re-announced with a `Commit_Message`. A report with `state = .Chosen` came from a peer's decided cell, so it is recorded and announced as a commit. A report with `state = .Voted` is re-proposed under the leader's ballot with `send_accept`: the highest-vote rule applied to that slot. No report at all means the no-op is proposed. Each `send_accept` records the leader's own vote durably (a `Write_Vote`) before the `Accept_Message` leaves.

> **Warning: A no-op is still a real value**
> It must be comparable, self-contained, and harmless when applied. Its type is the host's `Value`; the protocol has no special no-op tag. If the state machine cannot apply the no-op, it cannot apply the log.

When the window cannot yet hold the whole chunk because the memory floor is too far behind, `resolve_chunk` proposes as much as fits and reports the chunk unresolved. The node stays in `.Preparing`, and `tick` retries `maybe_resolve_chunk` until the host advances the floor or the election times out.

3.6 A Worked Recovery: Slots 10, 11, and 12

Three voters, ids 1, 2, and 3; both quorums are majorities of two. Node 1 led under ballot b_1 and delivered through slot 9. It proposed `X` in slot 10, and node 2's `Accepted` gave it a write quorum, so node 1 committed slot 10 and sent a `Commit`; the `Commit` reached node 2 but not node 3. Node 1 proposed `Z` in slot 12, and that `Accept` reached node 3 only. Nothing was ever sent for slot eleven. Then node 1 crashed. Node 3's election timer fires, and the host calls `campaign(&node3, noop, &effects)`. Its `delivered_through` is 9.

Step	Actor	Event and reason
1	Node 3	Chooses a round above b_1, writes its own promise, and after persistence sends `Prepare{first = 10, scope = .Global}`, with `last` closing one chunk, to nodes 1, 2, and 3. Node 1 is down and never answers.
2	Node 3	Steps its own `Prepare`. Its promise is already durable, so it only sends itself `Promise{slot 12, vote b_1, state .Voted, Z}` and a manifest with `reported = 1`, `chosen_through = 9`, `more = false`.
3	Node 2	Writes a promise. Its cell 10 holds the decision (it voted, then received the `Commit`), so it sends `Promise{slot 10, state .Chosen, X}` and a manifest with `reported = 1`, `chosen_through = 10`. The manifest overtakes the promise on the wire.
4	Node 3	Receives node 2's manifest: `expected_in_range = 1`, `received_in_range = 0`. Node 2 is described but not complete. `complete` is 1, below the read quorum of 2. Nothing else happens.
5	Node 3	Receives node 2's promise for slot 10. `received_in_range` becomes 1; node 2 is complete. `complete` is 2. `quorum_fences` returns a chosen fence of 10 from node 2. `resolve_chunk` starts at slot 11.
6	Node 3	Slot 11 has no recovered report: `send_accept` proposes `noop` there. Slot 12 has the recovered vote `(b_1, Z)`: `send_accept` re-proposes `Z` under the new ballot. Both votes are written (`Write_Vote`) before the `Accepts` go out. Because the chosen fence is above its own `delivered_through`, it sends `Learn{from_slot = 10}` to node 2.
7	Node 3	`become_leader`: `next_slot` and `leader_base` become 13, the slot after the highest slot it knows. The role is `.Leader`.

Step	Actor	Event and reason
8	Node 2	Answers the `Learn` with `Commit{10, X}` and the two `Accepts` with `Accepted` for slots 11 and 12, each after writing its vote.
9	Node 3	Records the commit for slot 10 and reaches a write quorum of 2 for slots 11 and 12. It commits both, broadcasts `Commits`, and releases `X`, `noop`, `Z` in that order. `delivered_through` is 12 and `is_leader_caught_up` is true.

Notice what did not happen. Slot 10 was never re-proposed: the chosen fence put it out of reach, and the leader learned it instead. Node 2's recovered report for slot 10 sat unused, because `resolve_chunk` began above the fence. Had node 2 voted for `X` but never received the `Commit`, it would have reported `chosen_through = 9` and `Promise{slot 10, vote b_1, state .Voted, X}`, and the leader would have re-proposed `X` in slot 10 under its own ballot, which is the highest-vote rule at work. The `.Chosen` branch is for a peer that holds the decision but has not delivered through it, for instance one that received the `Commit` for slot 10 while slot 9 was still missing. Every path ends the same way: `X` is the only value slot 10 can ever hold.

3.7 The Stable-Leader Pipeline

After `become_leader`, `propose(&node, value, &effects)` assigns `next_slot`, advances it, and calls `send_accept`. Nothing waits for the previous slot to commit: the leader may have many slots in flight, and acceptors vote on each independently. `propose_batch(&node, values, slots, &effects)` does the same for up to `CHUNK_SLOTS` values in consecutive slots and one effect batch, returning the assigned slots in the caller's `slots` buffer; more values give `.Batch_Too_Large`, none give `.Empty_Batch`.

Pipelining needs a bound, or one slow follower would let the leader's memory grow without limit. The bound is the window:

src/consensus.odin

```
    if node.next_slot == max(Slot) do return 0, .Global_Slot_Exhausted
    if node.next_slot - node.memory_floor > Slot(W) do return 0, .Window_Full
```

`W` is the node's `WINDOW_SLOTS` parameter. `memory_floor` is the greatest slot the host has told the node it may forget. It rises only when the host calls `advance_memory_floor(&node, through)` after durably consuming every released entry through `through`:

src/node.odin

```
node_advance_memory_floor :: proc(node: ^Node($V, $M, $W, $C, $G), through: Slot) -> Error
{
    if through > node.delivered_through do return .Invalid_Slot
    node.memory_floor = max(node.memory_floor, through)
    return .None
}
```

Asking to advance past `delivered_through` is `.Invalid_Slot`, and the floor never moves down. `.Window_Full` is flow control, not a log limit: the proposal is refused, nothing is written or sent, and the same call succeeds once the floor moves.

Physically, slot `s` lives in the ledger cell `cell_of(s, WINDOW_SLOTS)`:

src/ballot.odin

```
// The window index of a slot. WINDOW is a power of two, so this is one mask.
cell_of :: #force_inline proc(slot: Slot, $WINDOW: int) -> int {
    return int((slot - 1) & Slot(WINDOW - 1))
}
```

`node_init` asserts at compile time that `WINDOW_SLOTS` is a power of two, so the ring index is `(s - 1) & (WINDOW_SLOTS - 1)`, one mask rather than a division. The cell is tagged with its slot number in the ledger's `slot` column. `claim_live` decides whether a cell may be taken for a new slot. The live window is $(\texttt{memory\_floor}, \texttt{memory\_floor} + W]$: a slot below it has been consumed by the host, and a slot above it would land in the cell of a slot that is still live, so both are refused, and live slots and cells stay in bijection:

src/consensus.odin

```
    if slot <= node.memory_floor || slot - node.memory_floor > Slot(W) do return 0, false
    l := &node.ledger
    cell := cell_of(slot, W)
    held := l.slot[cell]
    if held == slot do return cell, true
    if held == 0 || (held <= node.memory_floor && l.state[cell] == .Chosen) {
        ledger_open(l, cell, slot)
        return cell, true
    }
    return cell, false
```

A cell is retagged (`ledger_open` clears everything but the value storage) only when it is empty or when its old occupant is both at or below the memory floor and `.Chosen`. A cell holding a `.Voted` but undecided vote is never evicted, because that vote may be part of a quorum some future leader must discover. The tag makes a stale cell impossible to mistake for the slot that now maps to it.

> **Checkpoint: Window arithmetic**
> With `WINDOW_SLOTS = 8`, the host has advanced the floor to 40 and the leader's `next_slot` is 48. Is the next `propose` accepted? Compute `next_slot - memory_floor` and compare with the window before answering. Then say what `advance_memory_floor(&node, 41)` changes.

3.8 Membership Changes: The Stop Sign

A configuration is a fixed voter set with fixed quorum sizes. To change it, the log carries a stop sign: a special entry that names the next configuration and its members. The `Replicated_Log_Node` in `src/replicated_log.odin` layers this on the core `Node`; its entries are an `Entry` union of the host's `Value` and a `Stop_Sign`. From the moment a stop sign is pending on a node, `log_propose` refuses commands with `.Log_Sealed`; once the stop is decided in slot s, no slot above s is ever released to the application in that configuration, and the next configuration starts at $s + 1$ on the same slot line. The full treatment, including how a delayed message from the old configuration is rejected, is in the advanced-features chapter.

3.9 Global Slots on One Line

Application slot numbers form one increasing sequence. `Slot :: u64` counts from 1 and never resets: not on a new leader, not on a trim, and not on a configuration change. A decision above a stop sign may be abandoned and its position decided by the next configuration, but that position was never released in the old configuration. What the window bounds is residency, not history: at most `WINDOW_SLOTS` slots live in protocol memory at a time, and everything below the memory floor survives only in the host's journal and materialized state.

Slot arithmetic saturates rather than wrapping. Chunk limits, batch sizes, and continuation slots go through `slot_add`, which adds `min(offset, max(Slot) - slot)`, so no calculation can produce slot zero or a small slot from a large one. The only terminal condition is `.Global_Slot_Exhausted`, returned by `propose`, `propose_batch`, `campaign`, and chunk resolution when the next slot would be `max(Slot)`. It is a stop, not a rollover: restarting the counter in the same log would reuse consensus instances that acceptors may still hold votes for.

> **Exercise 11.1**

A leader crashes after slot 10's Accept reached one acceptor and slot 12's Accept reached a different one; nobody voted in slot 11. Describe what the next leader's phase one learns, and what it proposes in slots 10, 11, and 12.

Hint: Ask which acceptors are in the read quorum, what each one's manifest says, and which branch of `resolve_chunk` each of the three slots takes.

Review & Discussion

Reconstruct the phase-one recovery process for a newly elected leader:

- How `recover_base` and `recover_last` delineate the active recovery window.
- How individual `Promise_Message` entries and `Promise_Range_Message` manifests are combined into `Election_Peer` tracking structures.
- The precise condition under which a read quorum is satisfied and `resolve_chunk` freezes candidate proposals.
- Why a ledger cell holding an undecided vote can never be retagged or overwritten until it is decided and durable.

4 The Safety Argument

Learning Objectives
After completing this chapter, you will be able to:
- Formulate the asynchronous consensus model as five formal system axioms.
- Define "chosen" rigorously in terms of acceptor state and write quorums.
- Reproduce the inductive proof of the Synod agreement theorem from ballot invariants B1, B2, and B3.
- Trace how real-world engine adaptations (chunked recovery, sliding memory windows, rotating ownership, durability ordering, and reconfiguration stop signs) preserve mathematical safety.
- Map each formal safety lemma directly to the Odin procedure that discharges its premises.

Parts I to III told the story: an empty ledger, a quorum, a ballot, three rules, and a log of decrees. This chapter is the argument in its compact form. It states the model as axioms, defines the words the theorem uses, proves the Synod theorem for one decree in the style of Lamport's Theorem 1, extends it to the multi-decree log, and then walks through every place where `paxos-odin` does something Lamport's parliament did not. Every lemma names the procedure in `src/` that discharges its premise, and the closing table maps each obligation of the package documentation in `src/paxos.odin` to its lemma, its procedures, and the test or simulator oracle that exercises it. Where the code guarantees less than a sentence in the package documentation suggests, the lemma says exactly what is guaranteed and what the host must add.

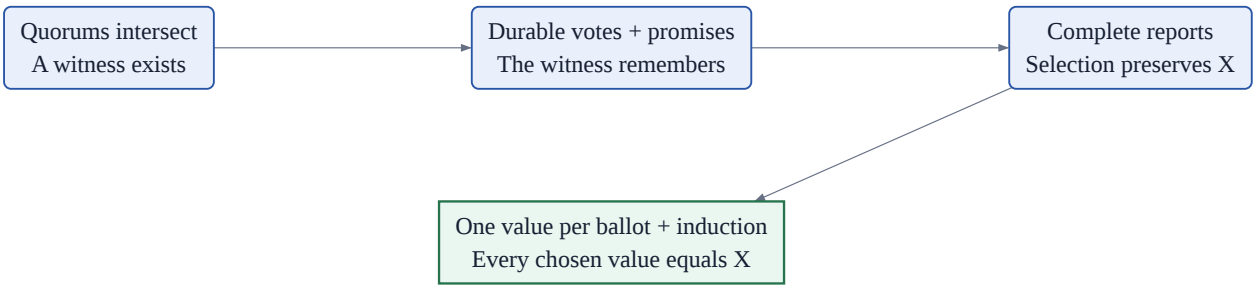


Figure 9: Read the proof as a chain of obligations. Intersection supplies a witness; persistence and promises preserve its evidence; recovery carries that evidence forward. The diagram is a map of the argument, not a substitute for its premises.

4.1 Axioms of the model

- **A1 (Processes).** A configuration has a fixed, finite set of members. A process runs the library's transitions one at a time and may crash at any instant. A crashed process may restart, and when it does it holds exactly the records it persisted before the crash and nothing else: every field of `Node` outside the `Ledger` is volatile and is reset or rebuilt by `node_restore`, using the ledger and the host's durable consumed floor and configuration.
- **A2 (Channels).** Envelopes may be lost, duplicated, reordered and delayed without bound, but never forged or corrupted. Every envelope a node processes was sent by `envelope.from` with the contents it carries; the host authenticates the transport, and `node_step` refuses an envelope whose `from` is outside the membership with `.Not_Member` and one not addressed to the node with `.Wrong_Recipient`.
- **A3 (Quorums).** A read quorum Q_1 is any set of `read_quorum_size` members and a write quorum Q_2 any set of `write_quorum_size` members, and `read_quorum_size + write_quorum_size > count`. `membership_init` refuses any other pair with `.Non_Intersecting_Quorums`, and `replicated_log_init_from_stop` builds the next configuration's membership through the same procedure.
- **A4 (Durability).** The host appends every `Write` of a transition to stable storage in the order given, syncs, and only then calls `confirm_writes_durable`; no message of that transition leaves before then, with the single exception of Lemma 10, and no later transition runs before then (`effects_reset` stops the process otherwise). A confirmed record survives every crash. On restart the host folds its journal in order through `ledger_replay_fold` (or `ledger_apply`) and calls `node_restore` with the slot through which it has durably consumed the log; that

consumed floor is itself durable and never decreases. A `Committed` entry is applied only after the batch that released it is confirmed.

- **A5 (No Byzantine behaviour).** Every process runs the library's code on its own ledger, never confirms a write that failed, never edits a ledger by hand, and reports its state truthfully in every message.

4.2 Definitions

Definition: D1: Ballot
A ballot is one integer, `Ballot :: distinct u64`, built by `ballot_make(round, priority, node)` as `round << 24 | priority << 16 | node`. The three fields occupy disjoint bit ranges (40, 8 and 16 bits), so distinct triples give distinct integers and integer order is the lexicographic order on $(\text{round}, \text{priority}, \text{node})$. The **proposer** of b is `ballot_node(b)`; a **campaign ballot** has `ballot_round(b) >= 1`; the **round-zero ballot** of a member n is `ownership_ballot(n)`, that is `ballot_make(0, 0, n)`. $b' > b$ means integer comparison.

Definition: D2: Vote
Acceptor a **votes** (b, v) in decree s when it records `Write_Vote{ballot = b, slot = s, value = v}`; afterwards the ledger cell for s has `vote_ballot = b`, `value = v` and state `.Voted` (or `.Chosen`, which keeps the vote). "a voted (b, v) in s" means such a record was confirmed durable at some time.

Definition: D3: Promise
The ledger holds a global promise `promised` (Lamport's maxBal, written by `Write_Promise`) and a per-decree promise `promised_at[c]` (written by `Write_Promise_At`, or implied by a vote). The **effective promise** of a for decree s is $p_a(s) = \max(\text{promised}_a, \text{promised\_at}_a[s])$, which is exactly `ledger_promise_for`.

Definition: D4: Chosen
Value v is **chosen** in decree s at ballot b when some write quorum Q_2 exists such that every $a \in Q_2$ voted (b, v) in s. Nobody has to know. v is chosen in s when it is chosen at some ballot.

Definition: D5: Decided
A node has **decided** (or **committed**) s when `record_commit` has recorded `Write_Chosen{slot = s, value = v}`; the cell is in state `.Chosen` and `ledger_chosen_at` returns v.

Definition: D6: Applied
The host has **applied** s once it consumed the `Committed` entry for s from `committed_slice`. `delivered_through` is the greatest slot released so far, and the host reports what it has durably consumed through `advance_memory_floor`.

The words are ordered by strength: applied implies decided on that node, decided implies chosen (Corollary 1), and chosen is the property the theorem is about. Chosen is a fact about durable votes and nothing else; no message, timer or leader is mentioned in D4.

4.3 The Synod theorem for one decree

Fix one decree s throughout this section.

Lemma 1 (quorum intersection). Every read quorum meets every write quorum: $Q_1 \cap Q_2 \neq \varnothing$.

Proof. $|Q_1 \cap Q_2| \geq |Q_1| + |Q_2| - N$, and A3 gives $|Q_1| + |Q_2| > N$. The inequality is `read + write <= total` in `membership_init`, refused with `.Non_Intersecting_Quorums`. ■

Lemma 2 (a vote respects the promise). If a votes (b, v) in s then $b \geq p_a(s)$ at the moment of the vote, and $p_a(s) \geq b$ ever after. $p_a(s)$ never decreases.

Proof. Every vote is recorded by one of two procedures. `on_accept` replies `send_nack` and records nothing when `msg.ballot < l.promised` or `msg.ballot < l.promised_at[cell]`; when it does record, it first sets `l.promised_at[cell] = msg.ballot`. `send_accept`, the proposer's own vote, returns without voting when `ballot < ledger_promise_for(l, cell)` and otherwise raises `promised_at` to the ballot. Promises only grow: `on_prepare` and `on_heartbeat` nack a ballot below `promised` and assign only a ballot at or above it; `promise_bounded` refuses the whole range if any `promised_at` exceeds the ballot; `ledger_apply` returns `.Promise_Regression` for a lower promise or a vote below a promise, and `ledger_replay_fold` folds with `max`. Across a restart the ledger is the replayed journal, so the same fields carry the same values (Lemma 9). ■

Lemma 3 (one value per ballot). For each ballot b at most one value is ever carried by an `Accept_Message` for (b, s), and every vote at b in s is for that value.

Proof. The proposer of b is `ballot_node(b)`, and a node builds ballots carrying only its own id (`start_campaign`, `start_revocation`, `ownership_ballot`). For a campaign ballot, the round is `greatest + 1` where `greatest` covers the node's own `ballot`, every round it observed, and `ledger_highest_ballot` (the global promise, every per-decree promise and every vote in the ledger); so each campaign of a node uses a round it never used before (a restart is Lemma 10). Within one campaign, a value enters s only through `send_accept`, which refuses to re-vote at the same ballot for a different value in a cell it is driving (`.Conflicting_Value`), and fresh proposals take `next_slot`, which `become_leader` sets above every used slot. For the round-zero ballot, `on_accept` discards any round-zero Accept unless `node.ownership` holds and `ballot_node(msg.ballot) == owner_of(node, msg.slot)`, and the owner proposes in an own slot once (Lemma 11). On the acceptor's side, `on_accept` answers a second value at the ballot it already voted with `.Conflicting_Value` rather than a vote, and `ledger_apply` does the same on replay. ■

Lemma 4 (B3, the max-vote rule). Suppose the proposer of a campaign ballot b issues an Accept for (b, s, v). Then there is a set Q_1 of at least `read_quorum_size` acceptors, each of which promised b for s before reporting, such that either (i) a received report supplies a vote (b'', v) in s whose ballot is at least as high as every vote reported by Q_1, or (ii) no received report supplies a vote in s, and v is the no-op or a fresh client value. The selected report in (i) may come from an additional peer whose chunk is not yet complete.

Proof. Every Accept at a campaign ballot is issued by `resolve_chunk` or, after `become_leader`, by `node_propose` and `node_propose_batch`. `resolve_chunk` runs only from `maybe_resolve_chunk`, which returns until `complete` reaches `membership_read_quorum`, where a peer is complete when its `Promise_Range_Message` arrived (`range_described`) and `received_in_range` reached the `reported` count it announced (`on_promise_range`, `on_promise`). Each such peer ran `on_prepare`, which promised the ballot (globally for `.Global`, per decree through `promise_bounded` for `.Bounded`) and then reported every used cell in the chunk above its trim anchor, one `Promise_Message` per cell, with the cell's `vote_ballot`, `state` and value. The candidate's `on_promise` keeps, per slot, the report with the greatest `vote`, promotes a `.Chosen` report over any vote, and returns `.Conflicting_Value` on two values at one ballot. `resolve_chunk` then proposes `recovered_value[cell]` for a slot with a `.Voted` record and `node.noop` for a slot with none, which is (i) and (ii) inside the chunk. Reports already received from additional peers can raise the selected ballot, but cannot lower it below the maximum from the complete quorum. `recovery_ready` freezes this selection before any phase-two vote and keeps it unchanged across a window-limited retry. A fresh value from `node_propose` lands at `next_slot`, which `become_leader` set above `ledger_highest_used` and both fences; by Lemma 6 no member of the final chunk's Q_1 holds a vote there, which is (ii). ■

Theorem 1 (Agreement). If v is chosen in s at b and w is chosen in s at b', then $v = w$.

Proof. Let b_0 be the least ballot at which any value is chosen in s, and let v_0 be that value; by Lemma 3 it is unique. It suffices to show that every Accept issued in s at a ballot $b' > b_0$ carries v_0: then a value chosen at b_0 is v_0 by Lemma 3, and a value chosen at $b' > b_0$ is the value of the Accepts its voters answered, again v_0.

Order Accept events in s by happens-before. Because every execution prefix is finite the order is well founded, and we argue by induction along it. Consider an Accept e for (b', s, u) with $b' > b_0$ and assume the claim for every Accept in s that happens before e.

The round-zero ballot of s is the least integer any acceptor will vote at in s (Lemma 11), so $b' > b_0$ is a campaign ballot, and Lemma 4 gives a read quorum Q_1 whose members promised b' for s before reporting. v_0 chosen at b_0 gives a write quorum Q_2 each of whose members voted (b_0, v_0) in s. By Lemma 1 pick $a \in Q_1 \cap Q_2$. By Lemma 2, a could not vote at $b_0 < b'$ after promising b' for s; so a voted (b_0, v_0) before it answered the prepare for b'. A cell's vote is replaced only by a vote at a greater ballot (`on_accept` records only at or above `promised_at`, which the earlier vote raised, and equality is the same value), and a vote leaves a ledger only for a slot at or below that node's delivered prefix or trim anchor (Lemma 8), in which case a's manifest fences s and e is never issued. So when a answered, it reported a vote (b_m, u_m) in s with $b_m \geq b_0$, and case (ii) of Lemma 4 is excluded.
By Lemma 4 (i), u is the value of a reported vote (b_M, u_M) at least as high as every vote reported by Q_1, with $b_M \geq b_m \geq b_0$. If $b_M = b_0$, then $u_M = v_0$ by Lemma 3. If $b_M > b_0$, the vote (b_M, u_M) was cast on receipt of an Accept for (b_M, s, u_M) that happened before the report, hence before e; by the induction hypothesis $u_M = v_0$. In both cases $u = v_0$. If instead some member of Q_1 reported s as `.Chosen`, `resolve_chunk` records that decision and issues no Accept in s at all, and the decision is v_0 by Corollary 1 and the same induction. ■

> **Why the induction runs along time, not along ballots**
> Lamport's proof of Theorem 1 inducts on ballot numbers, because in his protocol a priest that has voted at b'' refuses to answer a `NextBallot` below b''. Here a vote raises only the per-decree promise `promised_at`, while a `.Global` prepare compares against `promised`; so an acceptor may report a vote at a ballot **above** the candidate's, and the candidate re-proposes that value at its lower ballot. That is still safe, and the induction along happens-before shows why: the higher vote was produced by an Accept that already carried v_0.

Corollary 1 (decided implies chosen). If any node has decided s with value v, then v is chosen in s once the transition's required writes are durable.
Proof. `record_commit` is reached from four places. `on_accepted` calls it only when `acknowledged[cell]` reached `membership_write_quorum`, counting each member once (`bit_set_insert` on `acknowledgements[cell]`), for the ballot and slot the node is driving (`lead_slot`, `lead_ballot`), and only when its own cell still holds a vote at that ballot (`.Missing_Proposed_Value` otherwise); each acknowledgement is an `Accepted_Message` sent by `on_accept` after recording a vote at that ballot (or for a cell already holding it), and by Lemma 3 all those votes are for the same value, so they form a Q_2. `send_accept` calls it directly only when the write quorum is one, where the proposer's own vote is a Q_2. `on_commit` and `node_learn_chosen` record a value another node had decided (A2, A5), and `resolve_chunk` records a value a reporting peer had decided; both carry an earlier decision. Following these reports backward through the finite execution reaches a quorum-backed decision; this induction is over decision events, including repeated reports, rather than over the number of distinct nodes. ■
Theorem 1 with Corollary 1 is what the simulator's `AGREEMENT` oracle checks: it records the first value it sees chosen in each slot, counting durable votes directly in `persist_sim_write` so that a value chosen by a quorum whose leader then crashes still enters the golden log, and it fails the run if any node ever decides another.

4.4 Extension to the multi-decree log

Lemma 5 (decrees are independent). The ledger state of decree s is the cell `cell_of(s, W)` while `slot[cell] == s`, and nothing about s is ever read from a cell tagged with another slot.
Proof. Every read of a cell goes through `ledger_cell`, `ledger_vote_at`, `ledger_chosen_at` or `claim_live`, each of which compares `l.slot[cell]` with the slot it was asked about. A cell is retagged only by `ledger_open`, reached from `claim_live` (live), `ledger_claim` (replay) and `ledger_clear_cell` (restart), and it clears the promise, vote and state as the tag changes. So a vote, promise or decision belongs to one decree, and the Synod argument applies to each decree separately; only the global `promised` is shared, and sharing a promise can only refuse more votes, never permit one. ■
Lemma 6 (one campaign covers every decree above its base). Let a candidate at campaign ballot b become leader. Then for every decree $s \geq$ `recover_base` there is a set of at least `read_quorum_size` acceptors that promised b before reporting their votes in s, and the reports for s were complete before the candidate proposed in s.

Proof. `start_campaign` sends `Prepare_Message{ballot = b, first = recover_base, last = recover_last}` with scope `.Global`. `on_prepare` answers only if `msg.ballot >= l.promised`, sets `promised = b`, and then reports every used cell with `first <= slot <= last` above its anchor; it sets `more` if it holds a used cell above `last`. A `.Global` promise covers every decree, so a member that answered the first chunk refuses every lower ballot in every later chunk as well. `maybe_resolve_chunk` proposes nothing until a read quorum of members is complete for the current chunk; then `resolve_chunk` drives the chunk and, if any complete member said `more`, `begin_next_chunk` resets every `Election_Peer` except its fences, clears `promise_seen`, and sends the next `Prepare` for `[last + 1, ...]` under the same ballot. Each chunk therefore has its own complete read quorum, and each of its members reported every vote it held in that chunk after promising b. When no complete member reports `more`, no member of that final read quorum holds a vote above `last`, which is what Lemma 4 (ii) needs for fresh proposals above the last chunk. ■

The per-chunk bookkeeping is what lets the candidate tolerate A2: a manifest that overtakes its promises, a duplicated promise, or a chunk answered in two halves changes `expected_in_range` and `received_in_range` and nothing else, and the test `review_promise_reordering_deduplication_and_validation` runs those orders.

Lemma 7 (holes). Filling a hole with the host's no-op is an ordinary proposal and cannot conflict with a chosen value.

Proof. A hole is a slot in the drive range with no recovered vote and no recovered decision; `resolve_chunk` calls `send_accept(node, slot, node.ballot, node.noop.?, effects)`, the same procedure as any proposal, and `maybe_resolve_chunk` refuses to resolve at all with `.Missing_Noop` if no no-op was recorded. Lemma 4 (ii) holds for the slot, so if any value were chosen there at a lower ballot, its Q_2 would meet the chunk's Q_1 in a member that voted before promising and reported it, a contradiction; and Theorem 1 covers every higher ballot. The no-op is a `Value` like any other, with no tag, so nothing downstream distinguishes it. ■

Lemma 8 (fences and the memory window). A vote or decision leaves an acceptor's ledger only for a slot at or below that acceptor's delivered prefix or trim anchor, and a candidate never proposes at or below the greatest delivered prefix or trim anchor any complete member reported.

Proof. Three procedures drop a slot from a cell. `claim_live` admits only slots in the live window $(\texttt{memory\_floor}, \texttt{memory\_floor} + W]$, so the map from live slots to cells is a bijection and no cell is ever tagged with a slot whose predecessor in that cell is still live; within the window it retags a cell only when `held <= node.memory_floor` and the cell is `.Chosen`; `node_advance_memory_floor` keeps `memory_floor <= delivered_through`. `node_resume_at` clears open votes at or below `max(floor, anchor.chosen_trim_slot)` and sets `delivered_through` to that value. `ledger_claim`, on replay, reuses a cell whose slot is `.Chosen` or at or below the anchor for a later slot, and the later record exists only because the live node had claimed the cell under the first rule. In each case the dropped slot is at or below `delivered_through` or the anchor of that node, and both travel in every `Promise_Range_Message` as `chosen_through` and `anchor`. `quorum_fences` takes the maximum over the candidate's own values and every `Election_Peer`; `resolve_chunk` starts driving at `max(recover_base, fence + 1)`, and `become_leader` sets `next_slot` above both fences. A slot at or below a delivered prefix is decided on that node, hence chosen (Corollary 1), and a slot at or below a trim anchor is chosen by the definition of `Trim_Anchor`; so fencing forgoes only proposals that Theorem 1 would have forced to carry the chosen value anyway. The leader learns the fenced slots through `request_learn` instead, and slots below its memory floor come from the host through `Serve_Range_Request`. ■

The fence is not an optimisation. Without it, an acceptor whose vote for a chosen slot had been cleared by `node_resume_at` could be the only member of $Q_1 \cap Q_2$, and the candidate would see a hole where a decision lies. `review_recovery_preserves_fences_across_chunks` and `review_snapshot_preserves_votes_above_anchor` exercise both fences.

4.5 Durability

Lemma 9 (indelible ink). No promise or vote is observable by another node before it is durable, and a restarted node's ledger contains every promise and vote it ever revealed. Hence Lemma 2 holds across restarts.

Proof. `on_prepare`, `promise_bounded`, `on_heartbeat`, `on_accept` and `send_accept` each call `effects_add_write` before `effects_add_message` in the same transition, and `effects_add_write` sets `writes_pending`. With the default `Durability_Gate.Enforced`, `effects_messages_slice` calls `host_order_violation`, which stops the process, while `writes_pending` is set, and so does `effects_reset`, so neither the messages nor the next transition can precede `confirm_writes_durable` (A4). A restart replays the journal: `ledger_replay_fold` keeps the greatest promise, every per-decree promise and every vote for a slot still resident, and `node_restore` installs the result. Volatile bookkeeping is not needed for Lemma 2: the promise and the vote are the ledger. A restart lowers no promise, and the only cells it clears are those Lemma 8 covers. ■

The simulator injects a crash at each of `Before_Writes`, `Partial_Writes` and `Partial_Messages` in `process_effects`, replays the journal into a fresh `Ledger` through `ledger_replay_fold`, restores with the consumed floor, and checks `PROMISE REGRESSION` and `VOTE BELOW PROMISE` on every record it persists.

Lemma 10 (the pre-durable exception). Let the host transmit, before its barrier, only the envelopes `pre_durable_next` yields: `Accept_Messages` whose ballot has round at least one. Theorem 1 continues to hold provided a proposer that crashes with such an Accept in flight campaigns after restart at a round greater than the round of that ballot. `start_campaign` and `start_revocation` guarantee this unconditionally: each records the proposer's own promise for the new ballot in the same batch as the `Prepare` it sends (`Write_Promise` from `start_campaign`; the `Write_Promise_At` records of `promise_bounded` from `start_revocation`), and A4 makes that batch durable before any `Prepare` leaves. The restarted ledger therefore always holds a record at the round of every ballot the proposer ever announced.

Proof. An Accept claims nothing about its sender's durability: it asks acceptors to vote, and each acceptor persists its own vote before answering (Lemma 9). The only record the crash can lose is the proposer's own vote at (b, s), and a lost vote shrinks the set of votes at b; a subset of a Q_2 is not a Q_2, so nothing chosen becomes unchosen. The proposer counts its own vote toward a quorum only in `on_accepted`, a later transition, which A4 forbids before the vote is confirmed. What Lemma 3 needs is that b is never used again in s for another value. `start_campaign` and `start_revocation` compute `greatest` from `ledger_highest_ballot`, which scans `promised`, every `promised_at` and every `vote_ballot`; so if any record at the round of b survived, the next ballot has a greater round and the old votes at b are ordinary votes at a lower ballot that Lemma 4 will report. A record always survives: the proposer's own promise for b was written before its `Prepare` left, and a `Prepare` precedes every Accept at b. ■

> **Warning: Why the proposer promises itself**
> The candidate's self-addressed `Prepare` is an ordinary envelope, and `maybe_resolve_chunk` counts a read quorum without requiring the candidate itself in it. If the candidate's promise were written only when that envelope came back, a candidate that reached its quorum from peers alone, proposed, and crashed before its first vote at b was durable would hold no record at that round, and its restart could build the same ballot again: two values at one ballot in one decree, and Lemma 3 gone. With majority quorums at most one of the two is chosen, but a later phase one may report both, `on_promise` counts a report before it returns `.Conflicting_Value`, and `node_tick` retries `maybe_resolve_chunk`, so a campaign could resolve on whichever report arrived first. This is why `start_campaign` writes `Write_Promise` for its own ballot in the batch that carries the `Prepare`, and why `start_revocation` runs `promise_bounded` on itself before broadcasting. The premise of this lemma is discharged by the library, not by a host rule.

The exception is unsound at round zero, and `pre_durable_next` withholds round-zero Accepts for that reason. An owner's ballot is the same integer in every own slot, so after a restart `next_usable_own_slot` would hand it the same slot again if its vote there were not durable, and it would propose a different value at the same ballot in the same decree. The simulator's vote oracle, which fails a run when one ballot accepts two values in one slot, found exactly that counter-example: an owner reused its ballot for a different value after a crash. With round-zero Accepts behind the barrier, the owner's vote is in `ledger_highest_used` before any acceptor can see the Accept, and `node_resume_at` sets `own_next` above it.

4.6 Rotating ownership

Lemma 11 (ballot partition). Under `rotating_ownership`, in every decree s the round-zero ballot `ownership_ballot(owner_of(node, s))` is the least ballot any acceptor votes at, it belongs to the owner alone, and every other ballot has round at least one; so B1 holds per decree and Lemma 3 applies.

Proof. `owner_of` is `membership_get((s - 1) mod count)`, a function of s and the fixed membership, so every node computes the same owner. `on_accept` discards a round-zero Accept unless `node.ownership` holds and `ballot_node(msg.ballot)` is that owner. The owner proposes in s through `propose_owned`, which takes `next_usable_own_slot` and then moves `own_next` to `own_slot_from(slot + 1)`, so it proposes once per own slot; after a restart `node_resume_at` recomputes `own_next` above `ledger_highest_used`, which holds the vote (Lemma 10). Every other ballot comes from `start_campaign` or `start_revocation`, whose round is `greatest + 1 >= 1`, and the round occupies the top 40 bits, so any such ballot exceeds any round-zero ballot. ■

Lemma 12 (a revocation is a phase one). A revocation at ballot b over the range $[f, l]$ satisfies the premises of Lemma 4 for every decree in the range, so Theorem 1 holds with round-zero and campaign ballots mixed in one decree.

Proof. `start_revocation` picks a fresh round and sends `Prepare_Message{scope = .Bounded, first = f, last = l}` with $l - f <$ `CHUNK_SLOTS`. `on_prepare` first nacks a ballot below the global promise, then calls `promise_bounded`, which walks the range twice: it returns false, and the acceptor sends nothing, if any slot above the memory floor has no live cell or a `promised_at` above b; otherwise it records `Write_Promise_At{ballot = b, slot}` for every slot in the range before a single `Promise_Message` leaves. So a member the candidate counts as complete has promised b for every decree of the chunk, which is all Lemma 2 needs in s. The reports and `maybe_resolve_chunk` are unchanged. `resolve_chunk` runs with `drive_all` set, so every decree of the range is settled: a recovered round-zero vote is re-proposed at b (B3, the test `ownership_revocation_keeps_a_seen_vote`), a recovered decision is recorded, a hole takes the no-op. The revoked owner is fenced by `promised_at`: `on_accept` nacks its round-zero Accept, and `next_usable_own_slot` steps over a slot whose effective promise exceeds `ownership_ballot(node.id)`. After the chunk `become_leader` returns the revoker to `.Follower`; there is no standing leader to reuse the ballot. ■

Lemma 13 (skips). A skip is a proposal of the no-op at the owner's ballot in an own slot.

Proof. `skip_idle_slots` calls `propose_owned(node, noop, effects)` for own slots up to `highest_seen`, at most `min(C, SKIP_BURST)` per tick; nothing distinguishes the resulting Accept from any other round-zero proposal. ■

Lemma 14 (resubmission). A resubmitted suggestion is a new proposal in a new decree and violates nothing.

Proof. `queue_resubmit` is called from two places, and both key on the ledger alone: `on_accept`, when a higher ballot's value is about to overwrite the cell's own round-zero vote, and `record_commit`, when the decided value differs from that vote. The vote's ballot identifies the suggestion, so the check survives a revocation the owner itself started (which clears the lead columns). A suggestion overwritten in `on_accept` may yet be chosen at the revoker's ballot only if the revoker re-proposed it (Lemma 4), in which case the resubmission decides it twice; that is the at-least-once the host already handles by command id. `drain_resubmits` proposes it through `propose_owned`, that is, at the owner's ballot in a later own slot; Lemma 11 applies to that decree. The queue holds one chunk; a burst beyond it is dropped for the host's ordinary retry, so the guarantee is at-least-once only through the host. ■

Liveness, informally. A stalled prefix does not stay stalled. `tick_ownership` counts `stall_ticks` while `delivered_through < highest_seen`, asks the owner of the stuck slot for what it knows every `heartbeat_interval_ticks`, and starts a revocation of the chunk above `delivered_through` at `election_timeout_ticks`; a revoker that times out revokes again at a higher round. A live owner either proposes in its slots or skips them (Lemma 13), `resend_to` retransmits open Accepts and decisions, and `emit_contiguous` releases the prefix as it fills. None of this is proved here; it is the subject of the simulator's `LIVENESS` and `CONVERGENCE` oracles under `--ownership`.

4.7 Stop signs

Lemma 15 (a decided stop sign seals its configuration). Let stop sign σ be chosen at slot s in configuration K. Then (i) no node whose ledger holds σ, voted or decided, admits a new entry into K through

`replicated_log_propose`, `replicated_log_propose_batch` or `replicated_log_propose_stop_sign`; (ii) no `Replicated_Log_Node` of K that has decided s ever releases, reports or reads a slot above s, so a value the core nevertheless chooses there is **abandoned**; (iii) the next configuration continues at $s + 1$ on the same slot line, and a message from K cannot act there.

Proof. (i) All three proposal procedures return `.Log_Sealed` when `replicated_log_is_sealed` holds, which is `stop_pending || stop_sign != nil`. `replicated_log_recalculate_stop_pending` runs after every transition (`replicated_log_observe_effects`) and after restore (`replicated_log_observe_durable`); it scans every used cell of the ledger for a `Stop_Sign` whose `configuration_id` exceeds the node's own, so a vote for σ seals the node as surely as a decision does. (ii) `replicated_log_observe_effects` first records the earliest decided stop sign and its slot through `replicated_log_observe_stop`, and then `replicated_log_abandon_above_seal` cuts `effects.committed` at the first slot above `stop_slot`. Releases are contiguous (Lemma 16), so s is released before or together with $s + 1$, and the seal is on record before the cut runs in the same call; after a restart `replicated_log_observe_durable` rediscovers the seal from the `.Chosen` cells before any transition runs. `replicated_log_decided_through` reports at most `stop_slot`, and `replicated_log_read` and `replicated_log_read_decided` refuse the first abandoned slot. (iii) `replicated_log_init_from_stop` builds the membership from σ through `membership_init` and calls `replicated_log_continue_at`, which starts an empty window with `memory_floor = delivered_through = s` and `next_slot = s + 1`, refusing an anchor above s with `.Trim_Regression`. `replicated_log_step_checked` compares `Log_Envelope.configuration_id` with the node's own before the core sees the envelope, and on a mismatch resets the effects and returns `.Configuration_Mismatch` with no write and no message. ■

> **Warning: Sealed at release, not at choice**
> The core can still **choose** a value above s in K. `resolve_chunk` re-drives a reported vote with no seal check, so a minority vote an earlier leader left above s can be chosen by a later one; and under rotating ownership an owner that has not yet learned of σ can have a suggestion decided in its own slot, which `reconfiguration_sim_ownership_abandons_decisions_above_the_seal` provokes on purpose. Such a value is chosen in the sense of D4 and abandoned by the log: never released, invisible to `committed_at` and `read_decided`, and decided afresh by the next configuration with its own quorums, following Lamport, Malkhi and Zhou's rule that the acceptors of an instance are those of the configuration that owns it. S1 therefore holds for everything the host ever sees. Two consequences remain the host's: a client value abandoned this way must be proposed again in the next configuration, and a host that inspects the core ledger through `replicated_log_ledger` sees the abandoned decision and must not act on it.

4.8 Contiguity

Lemma 16 (contiguous delivery). Every `Committed` entry a node releases is for slot `delivered_through + 1` at the moment of release, its value is chosen, and `delivered_through` then becomes that slot; so the sequence released by one node is $1, 2, 3, \ldots$ from its restart base, with no gap and no repeat.

Proof. `emit_contiguous` loops from `next = delivered_through + 1`, stops at the first cell that is not tagged `next` or not `.Chosen`, and for each released slot appends `Committed{slot = next}` and sets `delivered_through = next`. The one other release is the pass-through in `record_commit`, taken only when `claim_live` fails (the slot is just past the live window) and `slot == node.delivered_through + 1`; it records `Write_Chosen`, releases exactly that slot, advances `delivered_through` and then calls `emit_contiguous`. Record and entry both point at the single `pass_through` field, so `record_commit` takes this path at most once per transition: a second such decision in the same batch is dropped and learned again later, never released with the wrong value. Every released value is a `.Chosen` cell or a value `record_commit` was about to record, chosen by Corollary 1. After a restart `node_resume_at` sets `delivered_through` to the restart base, the host's consumed floor or the trim anchor, whichever is greater, so the host sees the same sequence continue; a slot delivered before the crash but not yet consumed is released again, which is why the host applies idempotently. The standalone `Learner` gives the same guarantee through `released_through` in `learner_learn_chosen`. ■

The simulator's `CONTIGUITY` oracle fails the run if any node releases a slot other than `consumed + 1`.

4.9 Chunk-local recovery scratch

The volatile recovery arrays and each peer's report bitmap have capacity `CHUNK_SLOTS`, independently of the durable ledger window. For an active range `[recover_base, recover_last]`, `recovery_index` first checks range membership, then checks `slot - recover_base < CHUNK_SLOTS`, and only then converts the offset to an array index. Subtraction is therefore nonnegative and the index is bounded. Distinct slots in the range have distinct offsets. This argument does not require the chunk size to be a power of two or to divide the ledger window.

`reset_recovery_chunk` invalidates slot tags, states, ballots, and per-peer duplicate bitmaps at a new campaign or chunk. Payload bytes need not be zeroed: they are read only through a valid state and matching absolute slot tag. Delayed reports are rejected by ballot and range before they can access a new chunk's scratch. The cross-chunk trim and chosen-prefix fences are retained separately.

Once enough complete peer manifests constitute a read quorum, `recovery_ready` freezes the collected selection and `recovery_more`. This happens before the first phase-two vote. A window-limited retry retains that same selection; a late promise cannot replace a value already issued under this ballot. New chunks clear the freeze. Without this rule, a late higher losing vote could replace a proposal made from an earlier complete read quorum during a partial drive.

Resolution copies selected payloads into ledger-owned storage before emitting borrowed writes and messages. No outgoing effect borrows scratch that will be invalidated by chunk rollover. The refactor changes volatile representation, not journal or wire formats, quorum intersection, or the persistence-before-reply rule.

Tests exercise absolute-slot reference selection, non-power-of-two chunks, ring crossings, stale and duplicate reports, a blocked partial drive, a late promise after that drive, and large payload lifetimes. The simulator additionally exercises small windows with both flexible-quorum extremes and crashes. These are checks of the implementation's correspondence to the argument, not a machine-checked proof.

4.10 Obligations, lemmas, procedures, evidence

Obligation	Procedures	Tests and oracles
B1 ballot uniqueness (Lemmas 3, 11)	`ballot_make`, `start_campaign`, `start_revocation`, `ownership_ballot`, `on_accept` (owner rule), `send_accept`	`test_ballot_ordering`, `ownership_three_owners_propose_concurrently`, simulator vote oracle "ballot accepted two values"
B2 quorum intersection (Lemma 1)	`membership_init`	`test_membership_validation`, `review_negative_quorums_and_learner_campaign`, `review_flexible_quorums_and_window_reuse`
B3 max-vote (Lemmas 4, 6, 7, 12)	`on_prepare`, `on_promise`, `on_promise_range`, `maybe_resolve_chunk`, `resolve_chunk`, `promise_bounded`	`test_lamport_b3_max_vote_rule`, `election_matrix_preserves_chosen_values`, `ownership_revocation_keeps_a_seen_vote`, `review_multichunk_recovery_and_retry_progress`, simulator `AGREEMENT` and `VALIDITY`
Votes respect promises (Lemma 2)	`on_accept`, `send_accept`, `ledger_apply`, `ledger_replay_fold`	simulator `PROMISE REGRESSION` and `VOTE BELOW PROMISE`, `node_assert_valid`
Decided implies chosen (Corollary 1)	`on_accepted`, `record_commit`	`review_duplicate_acknowledgements_do_not_make_a_quorum`, `review_missing_proposal_is_error`, `test_three_node_cluster_agreement`
Window and fences (Lemmas 5, 8)	`claim_live`, `ledger_claim`, `quorum_fences`, `resolve_chunk`, `become_leader`	`review_recovery_preserves_fences_across_chunks`, `review_snapshot_preserves_votes_above_anchor`, `review_replay_reuses_certified_trimmed_vote`

Obligation	Procedures	Tests and oracles
D1 indelible ink (Lemmas 9, 10)	`effects_messages_slice`, `effects_reset`, `effects_confirm_writes_durable`, `pre_durable_next`, `node_restore`	`test_effects_power_loss_barrier_flag`, `test_pre_durable_messages_iterator`, `test_host_managed_gate`, `test_node_restore_and_recovery`, simulator crash points and `CONVERGENCE`
Rotating ownership (Lemmas 11, 12, 13, 14)	`owner_of`, `propose_owned`, `skip_idle_slots`, `start_revocation`, `queue_resubmit`, `drain_resubmits`	`ownership_idle_owners_skip`, `ownership_revokes_a_crashed_owner`, `ownership_revoked_suggestion_is_resubmitted`, simulator `--ownership`
S1 stop-sign sealing (Lemma 15)	`replicated_log_is_sealed`, `replicated_log_propose`, `replicated_log_abandon_above_seal`, `replicated_log_step_checked`, `replicated_log_init_from_stop`	`test_replicated_log_stop_sign_seals_epoch`, `review_stop_seal_restore_and_completed_history`, `reconfiguration_cluster_handover_rejects_old_epoch`, `review_out_of_order_chosen_stop_blocks_proposals`, the `seal_expect_nothing_after` oracle of every `reconfiguration_sim_*` scenario
L1 contiguous delivery (Lemma 16)	`emit_contiguous`, `record_commit`, `learner_learn_chosen`	`test_learner_contiguous_release`, `review_leader_fetches_decisions_from_ahead_follower`, simulator `CONTIGUITY`

Exercise 12.1

Show where the proof of Theorem 1 uses A3 (quorum intersection), and what fails if read and write quorums may be disjoint. Then construct the failure concretely: five members, a read quorum of two and a write quorum of two, two candidates, one decree. Write down the votes each acceptor holds when the second candidate resolves its chunk and name the branch of `resolve_chunk` it takes.

Hint: The proof picks $a \in Q_1 \cap Q_2$ exactly once. If no such a exists, Lemma 4 case (ii) is no longer excluded, and `membership_init` is the only line between you and the no-op.

Review & Discussion

Rehearse the core structure of Theorem 1 using a single witness acceptor $a \in Q_1 \cap Q_2$:

- Why the ordering of a's vote relative to its subsequent promise prevents older values from being proposed.
- How `resolve_chunk` translates the witness's promise into the leader's phase-two proposal.
- The precise scope of Lemma 10 regarding the durability barrier and host obligations.
- How Lemma 15 enforces configuration isolation across reconfiguration stop signs.

PART IV

The Odin library

The proof becomes a bounded value in memory. The host owns the disk, the network, the clock, and the application state. The boundary between them is a small set of procedures and one caller-owned batch of effects.

5 Bounded Core State Machine

Learning Objectives
After completing this chapter, you will be able to:
- Instantiate a statically sized `Node` and its matching `Effects` buffer without dynamic heap allocation.
- Inspect Lamport's acceptor variables directly from the columnar `Ledger` data structure.
- Execute consensus transitions and consume emitted effects according to the strict persist-before-send contract.
- Manage borrowed value pointer lifetimes safely before subsequent mutations invalidate ledger storage.
- Recover node state deterministically from replayed `Write` records following process restart.
- Distinguish between recoverable `Error` return values, compile-time assertions, and fatal invariant stops.

5.1 Design Philosophy: Consensus Without I/O

The library performs no I/O and owns no threads or clocks. Every public transition has the same shape: it **mutates the node in place** and **fills a caller-owned `Effects` value** with what the host must now do. There is no callback, no socket handle, no allocator, and no wall-clock read in `src/`.

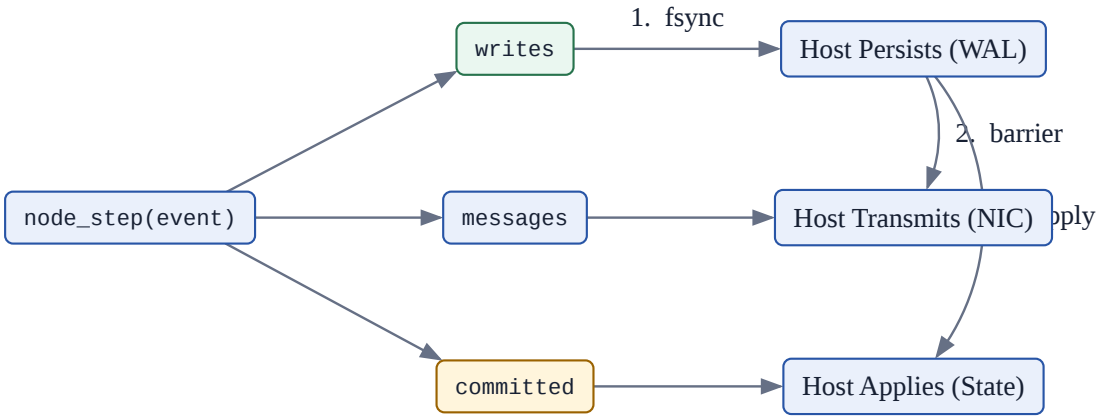


Figure 10: One transition consumes an event and fills three lists in a caller-owned batch. The host persists the writes, transmits the messages, and applies the committed entries, in that order.

A second host now exists, in another language. The Python package described in Part IX drives exactly this contract across a C boundary: it copies the writes, persists them, confirms, and only then reads the messages and the released entries. Its bridge compiles the core with the host-managed gate and enforces the same order itself, because the enforced gate ends the process and a host inside an interpreter cannot act on that. Nothing about the contract changes; only who is keeping it.

The reason is determinism. Given the same node state and the same event, a transition produces the same mutation and the same batch, whether it runs under `odin test`, inside the seeded fault simulator in `sim/`, or behind a real journal and a real socket. A failing simulation prints its seed and can be replayed step for step, because nothing inside a transition depends on time, memory layout, or scheduling. A transition is not a pure function, though, and the in-memory node is not a substitute for the writes it returns: when a call mutates `node.ledger`, it appends the matching `Write` record to the batch, and the node's memory is only a cache of what the journal will hold once the host has appended and synced those records.

5.2 Defining the Configuration

Two parametric structs carry the same five parameters. The first is the node.

src/node.odin

```
// One Multi-Paxos participant: proposer, acceptor, and learner in one bounded value.
// Every array is indexed by window cell or by stable member index; there is no
// per-slot struct, so a scan touches only the columns it needs.
//
// The host persists only the ledger, and only through the Write records it receives.
Node :: struct(
    $Value: typeid,
    $MAX_MEMBERS: int = DEFAULT_MAX_MEMBERS,
    $WINDOW_SLOTS: int = DEFAULT_WINDOW_SLOTS,
    $CHUNK_SLOTS: int = DEFAULT_CHUNK_SLOTS,
    $GATE: Durability_Gate = .Enforced,
) where intrinsics.type_is_comparable(Value) {
    id:                                 Node_Id,
    membership:                         Membership(MAX_MEMBERS),
    ledger:                             Ledger(Value, WINDOW_SLOTS),
```

The defaults live in `src/paxos.odin`: `DEFAULT_MAX_MEMBERS` is 7, `DEFAULT_WINDOW_SLOTS` is 256, and `DEFAULT_CHUNK_SLOTS` is 64. `MAX_MEMBERS` bounds the voting membership, `WINDOW_SLOTS` is the number of slot cells kept in memory, and `CHUNK_SLOTS` is how many slots one phase-one round recovers at a time (the Multi-Paxos chapter). `GATE` selects whether the durability contract is checked at runtime; leave it at `.Enforced` unless you have read the host-managed section.

The second struct is the batch, declared with the same parameters as the node it serves. The compiler rejects a mismatch: the fixture `effects_must_match_node` in `tools/check_contracts.py` pairs `paxos.Node(u64, 1, 4, 1)` with `paxos.Effects(u64, 1, 4, 2)` and asserts that `paxos.campaign(&n, 0, &e)` fails to type-check.

src/effects.odin

```
// Caller-owned output of one transition. Declare it with the same parameters as the
// Node it serves; the compiler rejects a mismatch. The zero value is ready to use.
//
// Nothing in a batch owns a value: writes, messages, and committed entries point into
// the node's ledger. Capacities are per-transition maxima: one recovery chunk of votes,
// each possibly followed by a local decision, plus one promise; one chunk per peer plus
// prepares, heartbeats, and one catch-up request (an ownership tick proposes at most
// one chunk and retransmits only on a quiet tick); one window plus one pass-through
// entry of decisions.
Effects :: struct(
    $Value: typeid,
    $MAX_MEMBERS: int = DEFAULT_MAX_MEMBERS,
    $WINDOW_SLOTS: int = DEFAULT_WINDOW_SLOTS,
    $CHUNK_SLOTS: int = DEFAULT_CHUNK_SLOTS,
    $GATE: Durability_Gate = .Enforced,
) where intrinsics.type_is_comparable(Value) {
    writes:         small_array.Small_Array(2 * CHUNK_SLOTS + 1, Write(Value)),
    messages:       small_array.Small_Array(
        MAX_MEMBERS * CHUNK_SLOTS + 2 * MAX_MEMBERS + 1, Envelope(Value),
    ),
    committed:      small_array.Small_Array(WINDOW_SLOTS + 1, Committed(Value)),
    requests:       small_array.Small_Array(MAX_MEMBERS, Host_Request),
    // True while this batch holds writes the host has not confirmed durable.
    writes_pending: bool,
}
```

5.2.1 Compile-time contracts

Capacity mistakes are caught before the program exists. `node_init` opens with three `#assert` statements: `MAX_MEMBERS` must lie in `1..=65535`, `WINDOW_SLOTS` must be a positive power of two, and `CHUNK_SLOTS` must satisfy

`1 <= CHUNK_SLOTS <= WINDOW_SLOTS`. Each message carries a `Hint`: naming the fix, for example "Invalid window. Hint: WINDOW_SLOTS must be a power of two."
The window is a power of two because a slot's cell is one mask, `cell_of` in `src/ballot.odin`: `(slot - 1) & (WINDOW - 1)`. `MAX_SUPPORTED_MEMBERS` is 65535 because a member's index and the node field of a ballot are both 16 bits wide; per-member sets are the library's own `Bit_Set(MAX_MEMBERS)` from `src/bit_set.odin`, not a native `bit_set`, so the member count is not capped at a machine word. The `where intrinsics.type_is_comparable(Value)` clause on both structs rejects a value type the language cannot compare with `==`; the fixture `non_comparable_value` shows `paxos.Node(map[int]int, 1, 4, 1)` refused with a diagnostic naming the `where` clause. `tools/check_contracts.py` runs nine compile-fail fixtures: a zero window, a zero chunk, a window of 3 slots (`window_not_power_of_two`), a chunk larger than its window, zero members, a `Membership(65536)` (`too_many_members`), a zero learner window, the non-comparable value, and the mismatched batch.

> **API anchor: `Node(Value, MAX_MEMBERS, WINDOW_SLOTS, CHUNK_SLOTS, GATE)`**
> One participant with every role. `Effects` takes the same five parameters, and a node only accepts a batch of its own configuration. in `src/node.odin`

5.2.2 What a `Value` may be

The `where` clause on `Ledger`, `Node`, and `Effects` states the rule: `Value` must be comparable, because it is stored in the ledger's `value` column and compared with == to detect a conflicting vote or decision. Odin equality compares active union variants and string contents. A fixed-size struct of integers, such as the `Command` in `examples/counter.odin`, needs nothing more. Comparability alone does not make a value suitable for replication. A string compares by contents, but a pointer compares by address; that address has no shared meaning on another machine. Prefer self-contained values. If a value refers to external bytes, the host must define consistent equality and encoding, and keep those bytes alive and immutable for every reference held by the protocol.

5.3 The Ledger: Lamport's Variables in Columns

The acceptor state of **The Part-Time Parliament** is three variables per decree: the greatest ballot promised, the ballot of the last vote, and the value of that vote. The `Ledger` is those variables laid out as parallel arrays over the window, with a fourth column for the decision and two bitmaps for the scans that phase one and catch-up need:

src/ledger.odin

```
Ledger :: struct($Value: typeid, $WINDOW: int = DEFAULT_WINDOW_SLOTS)
    where intrinsics.type_is_comparable(Value) {
    promised:    Ballot,
    anchor:      Trim_Anchor,
    slot:        [WINDOW]Slot,
    promised_at: [WINDOW]Ballot,
    vote_ballot: [WINDOW]Ballot,
    state:       [WINDOW]Cell_State,
    value:       [WINDOW]Value,
    // Bitmaps over cells: `used` has a vote or a decision, `chosen` has a decision.
    used:        Bit_Set(WINDOW),
    chosen:      Bit_Set(WINDOW),
}
```

`promised` is Lamport's `maxBal`, one global promise that covers every decree at or above the base of the campaign that made it. `promised_at[c]` is a per-decree promise, used only by bounded prepares under rotating ownership; the effective promise of a cell is the larger of the two (`ledger_promise_for`). `vote_ballot[c]` and `value[c]` are `maxVBal` and `maxVal`. `state[c]` is a `Cell_State`: `.Empty`, `.Voted`, or `.Chosen`, and once a cell is `.Chosen` its `value` is the decision. `slot[c]` tags which slot owns cell `c`, so a reused cell is never mistaken for an older one. The anchor is the chosen-trim record covered in the next chapter:

src/ledger.odin

```
// The trim anchor: every slot at or below `chosen_trim_slot` is chosen and has been
// folded into a host state image identified by `trim_id`. The core compares anchors
// by identity; the host binds the image's checksum to the id itself.
Trim_Anchor :: struct {
    trim_id:          u64,
    chosen_trim_slot: Slot,
}
```

Four queries read the ledger without touching the node. `ledger_vote_at(l, slot)` returns `(Ballot, ^Value, bool)` for a cell in state `.Voted`; `ledger_chosen_at(l, slot)` returns `(^Value, bool)` for a `.Chosen` cell; `ledger_highest_used(l)` walks the `used` bitmap for the greatest slot held; and `ledger_highest_ballot(l)` returns the greatest ballot anywhere in the ledger, which a campaign must exceed. `node_ledger(&node)` hands the host a `^Ledger(Value, WINDOW_SLOTS)` for inspection.

5.4 The Effect Contract

A batch carries four kinds of output. Writes are the durable records:

src/ledger.odin

```
// The durable records the host journals, in order. Values are referenced, not copied:
// a pointer is valid until the next transition on the node that produced it, and a
// host persists every write before it runs another transition.
Write_Promise :: struct {
    ballot: Ballot,
}

// A promise for one decree only (a revocation under rotating ownership).
Write_Promise_At :: struct {
    ballot: Ballot,
    slot:   Slot,
}

Write_Vote :: struct($Value: typeid) {
    ballot: Ballot,
    slot:   Slot,
    value:  ^Value,
}

Write_Chosen :: struct($Value: typeid) {
    slot:  Slot,
    value: ^Value,
}

Write_Trim :: Trim_Anchor

Write :: union($Value: typeid) {
    Write_Promise,
    Write_Promise_At,
    Write_Vote(Value),
    Write_Chosen(Value),
    Write_Trim,
}
```

Messages are `Envelope(Value)` values, each with a `from`, a `to`, and a `Message(Value)` union holding one of the nine wire messages from the protocol chapter. The three that carry a value do so by pointer: `Promise_Message(V){ballot, slot, vote, state, value: ^V}`, `Accept_Message(V){ballot, slot, value: ^V}`, and

`Commit_Message(V){slot, value: ^V}`; `message_value(message)` returns that pointer and whether the kind carries one. Committed entries are `Committed(Value){slot, value: ^V}`, released in slot order without gaps. Requests are `Host_Request` values; today the union has one variant, `Serve_Range_Request{peer, first, count}`, which asks the host to serve history below this node's memory floor from its own journal or state image.

5.4.1 Values travel by pointer

Nothing in a batch owns a value. A `Write_Vote`, an `Accept_Message`, and a `Committed` entry all point at the `value` column of the ledger cell the transition just wrote. The rule is stated at the top of `src/messages.odin`:

src/messages.odin

```
// Wire messages. Every variant is small and fixed-size; a value travels by pointer.
// Outbound, the pointer refers into the sending node's ledger and stays valid until
// that node's next transition, so a host serialises before then. Inbound, the host
// points it at the decoded value for the duration of the `step` call. An in-process
// transport that queues envelopes copies the value at enqueue time, as a codec would.
```

A real host serialises each write and each message before it steps the node again, and the copy happens inside its codec. An in-process transport has no codec, so it copies the value itself when it queues an envelope. The counter example names that pair a `Packet`:

examples/counter.odin

```
// An envelope in flight. A message points at a value inside the sender's ledger, so a
// transport copies the value when it queues the envelope, exactly as a codec would.
Packet :: struct {
    envelope: paxos.Envelope(Command),
    value:    Command,
}

packet_of :: proc(envelope: paxos.Envelope(Command)) -> (packet: Packet) {
    packet.envelope = envelope
    if value, carries := paxos.message_value(envelope.message); carries do packet.value
= value^
    return
}

packet_envelope :: proc(packet: ^Packet) -> paxos.Envelope(Command) {
    envelope := packet.envelope
    #partial switch &m in envelope.message {
    case paxos.Promise_Message(Command): m.value = &packet.value
    case paxos.Accept_Message(Command):  m.value = &packet.value
    case paxos.Commit_Message(Command):  m.value = &packet.value
    }
    return envelope
}
```

`packet_of` copies the value out at enqueue time; `packet_envelope` points the message back at the packet's own copy for the duration of the `step` call. The same idiom, generic over the value type, is `Packet(V)` in `tests/harness.odin`. The one exception to "into the ledger" is a decision for the slot just past the window edge: `record_commit` releases it through the node's `pass_through` field, which is why the `committed` list has room for `WINDOW_SLOTS + 1` entries. The pointer rule is the same either way: valid until the node's next transition.

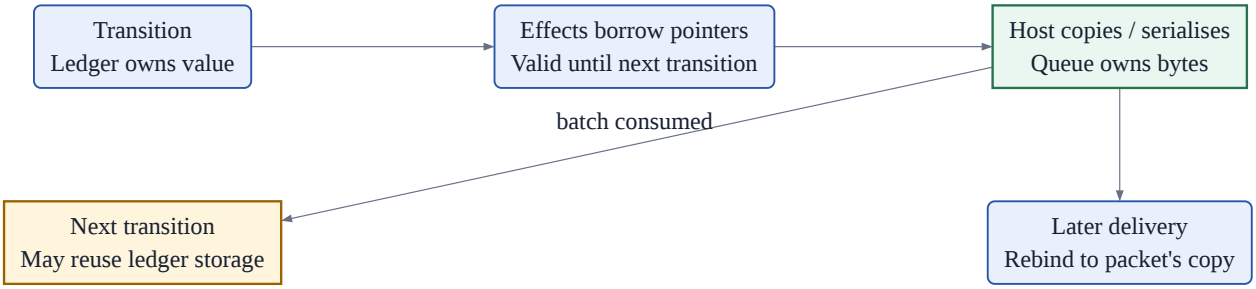


Figure 11: Borrowed pointers stop at the host boundary. Before another transition, the host consumes the batch, completes its required persistence, and copies or serialises anything needed later. An in-process queue must rebind each delivered envelope to its packet's own value, including when duplicating a packet.

5.4.2 The host commit sequence

The host consumes one batch in a fixed order. The counter's `host_commit` (the examples chapter) is this sequence with the journal left out:

```
err := paxos.step(&node, envelope, &effects)
if err != .None { /* see the section on errors */ }

// 1. Append every write, in slice order, copying each referenced value; then sync.
for w in paxos.writes_slice(&effects) do journal_append(&journal, w)
journal_sync(&journal)

// 2. Only now may anything leave the process.
paxos.confirm_writes_durable(&effects)

// 3. Transmit, copying each referenced value into the wire encoding.
for envelope in paxos.messages_slice(&effects) do transport_send(envelope)

// 4. Apply decided entries in slot order.
for entry in paxos.committed_slice(&effects) do apply(&state, entry.slot, entry.value^)

// 5. Serve history a peer asked for from below the memory floor.
for request in paxos.requests_slice(&effects) {
    switch r in request {
    case paxos.Serve_Range_Request: serve_from_journal(&journal, r.peer, r.first, r.count)
    }
}
```

`journal_append`, `journal_sync`, `transport_send`, `apply`, and `serve_from_journal` are the host's own procedures; the library supplies none. Two consequences of the order are easy to miss: a committed entry is applied only after its `Write_Chosen` record is durable, because step 4 follows step 2; and the next transition on the same batch calls `effects_reset`, so the whole sequence must finish, and every pointer must have been followed, before the node is stepped again.

Once the host has durably consumed a released prefix, it calls `paxos.advance_memory_floor(&node, through)`, which licenses reuse of the cells at or below `through`. A host that never advances the floor eventually receives `.Window_Full` from every proposal; that is backpressure, not a bug in the log.

5.4.3 The runtime gate

With the default `.Enforced` gate, reading the messages or resetting the batch while `writes_pending` is still true stops the process:

```
-- 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.
```

`host_order_violation` prints this, or the variant `reset discarded unconfirmed writes`, and calls `os.exit(1)`; it is declared `-> !`. The check is a `when G == .Enforced` branch, not a debug assertion, so it is present in optimised builds; `tools/check_contracts.py` builds each of its four durability fixtures with `-debug` and again with `-o:speed` and requires the named diagnostic and a `Hint:` line in both. Because every transition begins with `effects_reset`, a host that forgets `confirm_writes_durable` is stopped at its next call even if it never reads the messages.

The zero value of `Effects` is ready to use; the fixture `zero_value_is_ready` resets and reads a fresh batch with no initialisation call. Two procedures clear a batch. `effects_init` (`paxos.init(&effects)`) clears without checking the gate, for fresh memory or for a batch a crashed process can never complete; the simulator calls it after a simulated crash. `effects_reset` (`paxos.reset(&effects)`) clears after checking the gate, and every transition calls it on entry.

Two further queries help a host tune its storage. `requires_power_loss_barrier` is true when the batch carries a `Write_Promise`, a `Write_Promise_At`, or a `Write_Vote`; decision and trim records are derived state a host may persist with a cheaper barrier. `pre_durable_messages` returns an iterator whose `pre_durable_next` yields only `Accept_Message` envelopes whose ballot has a round above zero, the class the library documents as pipelinable before the local sync barrier completes. The comment explains the exclusion:

src/effects.odin

```
// Accept requests at a campaign ballot may leave before the local barrier: they ask peers
// to persist a vote and claim nothing about the sender's own durability, and a restarted
// proposer always campaigns at a fresh ballot. An owner's round-zero suggestion is the
// exception: its own vote is the only durable record that the instance was used, so it
// must wait for the barrier or a restart could reuse the ballot for another value.
```

The gate does not check this iterator, so a host that uses it takes on the reasoning itself.

> **Prediction Exercise**
> A transition returns `.None`; its batch holds one `Write_Vote` and two `Accept_Message` envelopes. The host calls `confirm_writes_durable`, sends both envelopes, and the machine loses power before the journal is synced. Which rule did the host break, and why could the library not catch it?

> **Warning: A failed sync invalidates the live node**
> The transition has already mutated the node. If the append or the sync fails, do not confirm the batch and do not keep stepping that node. Discard it, repair storage, replay only the records that did complete into a fresh `Ledger`, and restore a fresh node from that.

5.4.4 The host-managed exception

A host that groups several transitions behind one storage barrier may declare its node and batch with `GATE = .Host_Managed`. That removes the runtime check and transfers the obligation to the host. The rules are written on the type:

src/effects.odin

```
// .Host_Managed disables that check for hosts that own the durability boundary
// themselves, for example by grouping several transitions behind one storage barrier.
// Such a host must guarantee, by construction, that:
//   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.
```

```
Durability_Gate :: enum {
    Enforced,
    Host_Managed,
}
```

The spelling is deliberately searchable. For an example, read `paxos.Node(u64, 1, 64, 16, .Host_Managed)` and its `Effects` in `tests/test_durability.odin`. Search for `.Host_Managed` when auditing callers; each use takes on the same obligations.

5.5 The Public Transition Surface

`src/paxos.odin` groups each verb over its receiver types, so `paxos.propose` dispatches on whether the first argument is a `^Node` or a `^Replicated_Log_Node`. The table gives the `Node` signatures; parameter order matches the source.

Verb	Signature after the node	What it does
`init`	`(id, membership, options = {}) -> Error`	Voting follower at slot 1; `.Not_Member` if `id` is not in the membership.
`campaign`	`(noop, &effects) -> Error`	Starts phase one with a fresh ballot; `noop` fills recovered holes. `.Campaign_Disabled` when campaigning is off or ownership is on.
`propose`	`(value, &effects) -> (Slot, Error)`	Assigns the next slot on a leader, votes for it locally, and broadcasts `Accept_Message`.
`propose_batch`	`(values, slots, &effects) -> ([]Slot, Error)`	Up to `CHUNK_SLOTS` values into consecutive slots; `slots` is the caller's output buffer.
`step`	`(envelope, &effects) -> Error`	Processes one authenticated envelope addressed to this node.
`tick`	`(noop, &effects) -> Error`	Advances election, heartbeat, and resend counters by one interval.
`reconnected`	`(peer, &effects) -> Error`	A leader resends to the peer; a follower asks the peer to learn if it is the leader hint. `.Invalid_Peer` for the node itself.
`request_catch_up`	`(peer, from_slot, &effects) -> Error`	Sends `Learn_Message` for one chunk from `from_slot`.
`learn_chosen`	`(from, slot, value, &effects) -> Error`	Installs a host-certified decision on a non-voting node.
`set_campaign_enabled`	`(enabled)`	Turns campaigning on or off; a `.Preparing` node drops to `.Follower`.
`advance_memory_floor`	`(through) -> Error`	Licenses cell reuse through `through`; `.Invalid_Slot` above `decided_through`.
`install_chosen_trim`	`(anchor, &effects) -> Error`	Adopts a chosen trim anchor and emits `Write_Trim`.

Queries read plain fields and never touch the batch:

Query	Returns	Meaning
`role`	`Role`	`.Follower`, `.Preparing`, or `.Leader`.
`ballot`	`Ballot`	The ballot this node last campaigned with.
`id`	`Node_Id`	The local identity.
`current_leader`	`(Node_Id, bool)`	The leader hint, if any; not a lease.
`decided_through`	`Slot`	The greatest contiguous slot released to the host.
`leader_base`	`Slot`	The first slot this leadership assigned.
`proposal_frontier`	`Slot`	The slot the next proposal would take.
`committed_at`	`(Value, bool)`	A copy of one decided value still resident in the window.
`read_decided`	`(int, Error)`	Fills a caller buffer of `Committed(Value)` from `from_slot`; `.Trimmed` below the floor, `.Read_Buffer_Too_Small` if it does not fit.
`is_leader_caught_up`	`bool`	Every inherited slot is delivered; the doc comment adds "not a lease and not a read barrier".
`memory_floor`	`Slot`	The greatest slot whose cell the host released for reuse.
`trim_anchor`	`Trim_Anchor`	The adopted chosen-trim anchor.
`is_voting_member`	`bool`	False for a learner made by `init_learner`.
`is_campaign_enabled`	`bool`	Whether a timeout may start a campaign.
`ledger`	`^Ledger(Value, WINDOW_SLOTS)`	A pointer to the durable columns, for inspection.

> **API anchor: `paxos.step(&node, envelope, &effects)`**
> The one transition a transport calls. It rejects `.Wrong_Recipient` and `.Not_Member` before touching state, records the sender's decided prefix, and dispatches on the message variant. in `src/consensus.odin`

5.6 `Node_Options` and What Zero Means

`init` takes an optional `Node_Options` whose zero value selects every default, so `paxos.init(&node, id, membership)` and `paxos.init(&node, id, membership, paxos.Node_Options{priority = 2})` both read naturally.

src/node.odin

```
Node_Options :: struct {
    // Breaks ballot ties between rounds: higher wins. Zero is the lowest priority.
    priority:                          u8,
    // Follower ticks without leader contact before it campaigns. Zero means the default.
    election_timeout_ticks:            u32,
    // Leader ticks between heartbeat broadcasts. Zero means the default.
    heartbeat_interval_ticks:          u32,
    // Leader ticks between bounded retransmission scans. Zero means the default.
    resend_interval_ticks:             u32,
    // Refuse proposals with .Leader_Catching_Up until every inherited slot is delivered.
    gate_proposals_on_inherited_prefix: bool,
    // Start as an acceptor that promises and votes but never campaigns.
    campaign_disabled:                 bool,
```

```
    // Rotating slot ownership: every member proposes in its own slots without phase one.
    // Campaigns are refused; stalls are repaired by bounded revocations.
    rotating_ownership:                bool,
}
```

The timer defaults are 10 election ticks, 3 heartbeat ticks, and 10 resend ticks. A tick is whatever interval the host calls `tick` at; the library never reads a clock. `priority` is a `u8` because it occupies eight bits of the packed ballot; `rotating_ownership` selects the second way to lead, which has its own chapter.

5.7 Restoring State After a Crash

A restarted process has an empty `Node` and a journal of `Write` records. Two procedures fold records into a `Ledger`. `ledger_apply` is strict: a promise lower than the current one is `.Promise_Regression`, two values under one ballot and slot are `.Conflicting_Value`, a second value for a chosen slot is `.Conflicting_Commit`, a record addressing a cell still held by an earlier slot is `.Window_Overrun`, and an anchor that moves backward is `.Trim_Regression`; use it for one configuration's journal, where each of these is corruption. `ledger_replay_fold` is for a lifetime journal: promises fold to their maximum, a vote whose cell has since been claimed by a newer slot is skipped as dead history, and decisions and trim records keep the strict rules.

Because a `Write_Vote` or `Write_Chosen` carries a pointer, the journal must hold the value alongside the record and point the record back at it during replay. `tests/harness.odin` is the reference pattern:

tests/harness.odin

```
// One journaled record with its value copied out of the ledger.
Journal_Record :: struct($Value: typeid) {
    write: paxos.Write(Value),
    value: Value,
}

journal_append :: proc(journal: ^[dynamic]Journal_Record($V), writes: []paxos.Write(V)) {
    for w in writes {
        record := Journal_Record(V){write = w}
        #partial switch x in w {
        case paxos.Write_Vote(V):   record.value = x.value^
        case paxos.Write_Chosen(V): record.value = x.value^
        }
        append(journal, record)
    }
}

// Rebuilds a ledger from a journal with the lifetime fold.
journal_replay :: proc(
    journal: []Journal_Record($V),
    ledger: ^paxos.Ledger(V, $W),
) -> paxos.Error {
    for &record in journal {
        write := record.write
        #partial switch &x in write {
        case paxos.Write_Vote(V):   x.value = &record.value
        case paxos.Write_Chosen(V): x.value = &record.value
        }
        paxos.ledger_replay_fold(ledger, write) or_return
    }
    return .None
}
```

The simulator's restart path does the same fold inline and then restores every derived frontier:

sim/simulation.odin

```
    // Replay the lifetime journal into fresh durable state, then restore every derived
frontier.
    ledger: paxos.Ledger(u64, SIM_WINDOW)
    for &record in small_array.slice(&sim.journals[node_idx]) {
        write := record.write
        #partial switch &w in write {
        case paxos.Write_Vote(u64):   w.value = &record.value
        case paxos.Write_Chosen(u64): w.value = &record.value
        }
        sim_check(paxos.ledger_replay_fold(&ledger, write))
    }
    options := paxos.Node_Options{priority = u8(node_idx), rotating_ownership =
sim.config.ownership}
    floor := sim.consumed[node_idx]
    sim_check(paxos.restore(&sim.nodes[node_idx], id, sim.membership, ledger, floor,
options))
```

Four restore procedures differ in what they keep and what they clear:

- `restore(node, id, membership, ledger, floor = 0, options = {})` runs `init` first, so every volatile field starts fresh: role `.Follower`, zero ballot, no leader hint, no election state. It installs `ledger`, clears every cell at or below `max(floor, ledger.anchor.chosen_trim_slot)` that holds only an open vote, and resumes `memory_floor` and `decided_through` at that base. `floor` is the slot through which the host has durably consumed the log; zero for a node that never released anything.
- `continue_at(node, id, membership, floor, anchor, options = {})` starts an **empty** node on the same slot line with an inherited trim anchor, for a configuration handover or a state-image install; `.Trim_Regression` if the anchor lies above `floor`.
- `begin_recovery(node, anchor)` acts on a live node. It applies the anchor through `ledger_apply`, clears open votes at or below it, and drops the node to `.Follower` with its election state cleared. It keeps the promised ballot and every vote or decision above the anchor, because such a vote can belong to a chosen quorum this node has not yet learned about. It takes no batch, so the host persists the image and anchor itself before stepping the node again.
- `restore_learner(node, id, membership, ledger)` rebuilds a non-voting learner from its decision-only journal.

None of these restores application state, client sessions, or transport queues. The host restores its state image and applied slot from its own durable data and replays committed records above that slot idempotently before serving.

5.8 Errors

Every transition returns an `Error`; `.None` is zero, so `if err != .None` is the whole test. The enum in `src/errors.odin` groups its members as membership, input and addressing, role and capability, liveness and progress, durability and safety, and replicated-log and window errors. Not every error is retryable: `.Not_Leader` and `.Window_Full` describe a state the host can wait out, while `.Conflicting_Commit` and `.Promise_Regression` are safety incidents whose hints tell the operator to stop and keep the evidence. `explain_error` returns an operator-facing banner for each; the table `EXPLANATIONS` is indexed by the enum, so a value without an entry fails the exhaustive test in `tests/test_errors.odin`. This is the text for `.Not_Leader`:

```
-- NOT LEADER ------------------------------------------------------------------

This node has not completed phase one for its current ballot.
Hint: Route to current_leader() or wait for a successful campaign.
```

The counter example passes that string as the message of an `assert`; a host logs it and decides on retry or shutdown by the error's group.

Review & Discussion
Delineate the boundary between the consensus library and the host application:

- Assign data structures and obligations to their owner: `Node`, `Ledger`, `Effects`, network transport, disk journals, timers, and state machine application.
- Detail the exact ordering constraint that forbids dispatching outbound messages before disk sync completes.
- Explain how `confirm_writes_durable` enforces this barrier and what diagnostic triggers if violated.
- Specify the precise lifetime of borrowed value pointers emitted in an `Effects` batch.

6 Advanced Replicated Log Features

Learning Objectives
After completing this chapter, you will be able to:

- Drive consensus progression using logical ticks and explain the liveness guarantees of election and heartbeat timers.
- Distinguish election term bases from true leader leases and understand linearizable read requirements.
- Configure flexible asymmetric quorums and verify their compliance with the quorum intersection rule.
- Execute membership changes and epoch sealing using stop-sign reconfigurations.
- Catch up lagging followers and leaders using `Learn_Message` catch-up queries.
- Install trim anchors to compact log history without compromising uncommitted votes.
- Deploy non-voting `Learner` nodes for horizontal read-scaling.

6.1 Logical Time

The core owns no clock. Safety never depends on time, but liveness needs a way to suspect a silent leader, remind followers that a leader is alive, and resend what was lost. The host converts elapsed time into calls to `tick(&node, noop, &effects)` at whatever cadence it chooses, and three intervals in `Node_Options` are measured in those calls:

src/node.odin

```
    // Follower ticks without leader contact before it campaigns. Zero means the default.
    election_timeout_ticks:             u32,
    // Leader ticks between heartbeat broadcasts. Zero means the default.
    heartbeat_interval_ticks:           u32,
    // Leader ticks between bounded retransmission scans. Zero means the default.
    resend_interval_ticks:              u32,
```

src/paxos.odin

```
DEFAULT_ELECTION_TIMEOUT_TICKS   :: 10
DEFAULT_HEARTBEAT_INTERVAL_TICKS :: 3
DEFAULT_RESEND_INTERVAL_TICKS    :: 10
```

The zero value of `Node_Options` selects every default, so `paxos.init(&node, id, membership)` and `paxos.init(&node, id, membership, {election_timeout_ticks = 20})` both read naturally. A non-voting node returns from `tick` at once: it has nothing to suspect and nothing to resend.

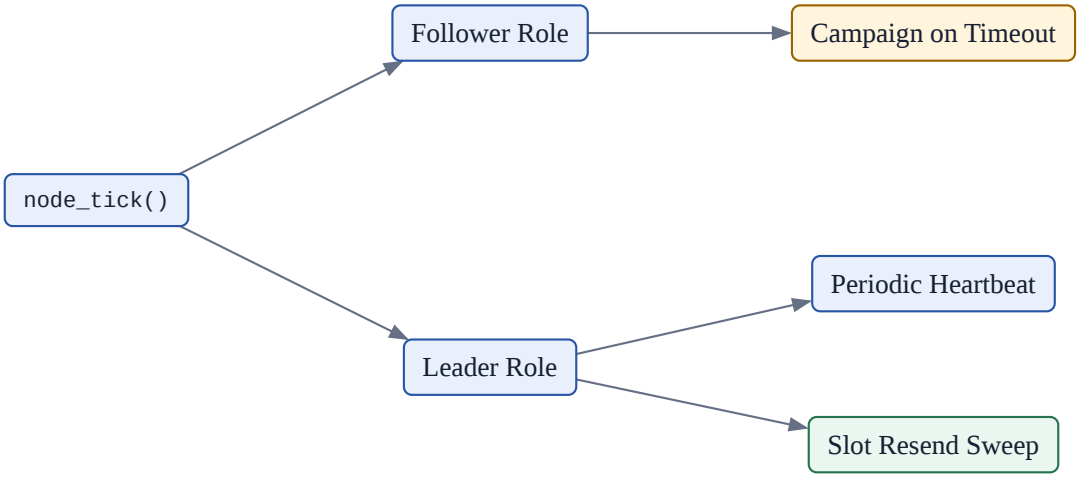


Figure 12: One logical tick advances all three counters, but only the current role's duties run: a follower may campaign, a leader may heartbeat and resend.

For a voter, one tick increments `election_ticks`, `heartbeat_ticks`, and `resend_ticks` (saturating, never wrapping), then acts by role:

1. A **leader** whose `heartbeat_ticks` reached `heartbeat_interval_ticks` broadcasts `Heartbeat_Message{ballot, decided_through}` to its peers. The heartbeat carries the leader's delivered prefix, so a follower can see that it is behind without any extra message kind.
2. A **leader** whose `resend_ticks` reached `resend_interval_ticks` calls `resend_to` for every peer. If the peer has reported a decided prefix above the leader's own, `resend_to` first sends it a `Learn_Message`, because leadership does not imply knowing every decision. Then it walks the ledger's `used` bitmap with `bit_set_next` from a per-peer cursor (`resend_cursor`), skipping cells at or below what the peer has already decided, resending a `Commit_Message` for a `.Chosen` cell and an `Accept_Message` for a `.Voted` cell this leader is driving under its current `lead_ballot`, and stops after `CHUNK_SLOTS` messages or one complete sweep. It wraps at most once and never visits the same used cell twice in that sweep. The cursor keeps its position, so a quiet peer cannot pin every retry to the first chunk.
3. A **candidate** still in `.Preparing` before its timeout retries `maybe_resolve_chunk`, for the case where the window could not hold the whole chunk the first time.
4. A **follower**, or a candidate whose election timed out, with `election_ticks` at or past `election_timeout_ticks` starts a campaign with the `noop` passed to this `tick`, provided campaigning is enabled. The timeout is suspicion, not proof: the old leader may be alive and partitioned.

A node with `rotating_ownership` set takes a different path, `tick_ownership` in `src/ownership.odin`, after the counters advance; the ownership chapter walks it. `election_ticks` resets whenever the node observes a leader (`observe_leader`): on a global `Prepare`, an `Accept` at a round above zero, a `Commit`, or a `Heartbeat`. The heartbeat handler has one more job:

src/consensus.odin

```
    // A heartbeat above the promise means this node missed the leader's prepare.
    // Promising is always safe, and it stops a needless election.
    if msg.ballot != l.promised {
        l.promised = msg.ballot
        effects_add_write(effects, Write_Promise{msg.ballot})
    }
    observe_leader(node, from, msg.ballot)
    if msg.decided_through > node.delivered_through do request_learn(node, from, effects)
```

A follower that never saw the winning `Prepare`, perhaps because it was partitioned during the election, adopts the heartbeat's ballot by promising to it, and the promise is written to `ledger.promised` and to the batch before anything else happens, exactly as in phase one. A heartbeat below the follower's promise gets a `Nack_Message` instead, which tells a stale leader to step down. `request_learn` asks for one chunk from `delivered_through + 1`.

API anchor: `tick`
`node_tick(node, noop, effects)` and `replicated_log_tick(node, noop, effects)`. The `noop` is stored for any campaign the tick starts; pass the same value the host passes to `campaign`. in `src/consensus.odin`

Prediction Exercise
A node just won an election. Its `leader_base` is 13 and its `delivered_through` is 9, because the chosen fence told it that a peer has decided through 12. It has `gate_proposals_on_inherited_prefix` set. A client proposes a value now. What does `propose` return, and what has to happen before the same call succeeds?

6.2 Taking Over: The Term Base

Winning phase one does not leave an idle log. The new leader is re-proposing recovered votes, filling holes with the no-op, and possibly learning slots below the chosen fence from a peer. New proposals go above all of that. `become_leader` records the boundary: it raises `next_slot` past the greatest used slot and both fences, and `leader_base` is the first slot this leadership may fill with a fresh value, while `next_slot`, reported by `proposal_frontier`, is the slot the next proposal would take. Right after the election they are equal.
A host whose values are independent of each other may pipeline straight through the takeover. A host whose values depend on applied state (a compare-and-swap, a hash chain, a sequence number derived from the last applied entry) cannot: the inherited slots below `leader_base` will decide ahead of any new proposal, and a value derived before they are applied is derived from the wrong state. Such a host delivers through `leader_base - 1` first. The option `gate_proposals_on_inherited_prefix` makes the core enforce this in `propose` and `propose_batch`, through the shared `proposal_gate`:

src/consensus.odin

```
@(private)
proposal_gate :: proc(node: ^Node($V, $M, $W, $C, $G)) -> Error {
    if !node.voting_member do return .Not_Voter
    if node.ownership do return .None
    if node.role != .Leader do return .Not_Leader
    if node.gate_proposals_on_inherited_prefix && node.delivered_through <
node.leader_base - 1 {
        return .Leader_Catching_Up
    }
    return .None
}
```

The option is off by default. With or without it, `is_leader_caught_up(&node)` reports whether `delivered_through` has reached `leader_base - 1`. This resolves the prediction: `propose` returns `.Leader_Catching_Up`, and it succeeds once the `Learn_Message` sent during recovery has been answered and slots 10 through 12 are delivered.

Warning: `is_leader_caught_up` is not a lease and not a read barrier
It reports prefix progress only: the leader has applied every slot it inherited. It says nothing about whether this node is still the leader, or whether a higher ballot has since chosen values this node has not seen. A linearizable read from local state additionally needs a quorum round trip or a read barrier through the log, driven by the host. Leader leases are a design proposal (POD 0004) and are not implemented in this library.

API anchor: `leader_base`, `proposal_frontier`, `is_leader_caught_up`
Three queries on `Node` and `Replicated_Log_Node`. The first two are slots; the third is `delivered_through >= leader_base - 1`. in `src/node.odin`

6.3 Flexible Quorums

Phase one needs a read quorum of promises; phase two needs a write quorum of votes. The single-decree argument only requires that every read quorum intersect every write quorum, so that a later ballot's phase one sees at least one vote from any earlier ballot's phase two. With uniform sizes over N members, that is

$$|Q_1| + |Q_2| > N.$$

Five voters	A	B	C	D	E
Write quorum: 2	vote X	vote X	-	-	-
Read quorum: 4	-	report X	report	report	report

B witnesses both quorums. Any four voters must include A or B.

Figure 13: Read and write quorums have different jobs. Here two votes choose X and four reports recover the past. Their overlap is forced by 4 + 2 > 5. The diagram assumes no intervening votes; the highest-vote argument handles that case.

Two write quorums need not intersect: within one ballot the leader proposes one value per slot, and across ballots the read quorum does the intersecting. The library validates the sizes when the membership is built:

src/membership.odin

```
// Validates and installs a membership. Zero overrides select majorities.
membership_init :: proc(
    m: ^Membership($MAX_MEMBERS),
    node_ids: []Node_Id,
    read_quorum_override: int = 0,
    write_quorum_override: int = 0,
) -> Error {
```

Zero means a majority. A size outside `1..=N` is `.Invalid_Read_Quorum` or `.Invalid_Write_Quorum`; a pair whose sum does not exceed N is `.Non_Intersecting_Quorums`, and the caller's membership is left untouched. For five voters:

N	Q_1 (read)	Q_2 (write)	What it buys and what it costs
5	3	3	Symmetric majorities. Any two voters may be down for both elections and commits.
5	4	2	A commit needs the leader plus one acceptor, so a stable leader tolerates three silent voters. An election needs four promises, so replacing the leader tolerates only one.
5	2	4	Elections need only two promises, but every commit needs four durable votes; two unavailable voters prevent a commit; one slow voter can be bypassed.
5	5	1	The leader commits on its own vote: `send_accept` calls `record_commit` before the `Accept` leaves. An election needs all five voters.

A smaller write quorum buys cheaper commits and pays for them at the next election. Choose the pair for the failure you expect to be common, and let `membership_init` check it:

```
m: paxos.Membership(5)
ids := [5]paxos.Node_Id{1, 2, 3, 4, 5}
err := paxos.membership_init(&m, ids[:], 4, 2) // Q1 = 4, Q2 = 2
```

Every `membership_*` query takes a `^Membership`. Up to `LINEAR_LOOKUP_LIMIT` (8) members, `membership_index_of` scans the members in order; above that it binary-searches them, which works because `membership_init` sorted them, so a membership near the 65535 bound is still one lookup per message.

6.4 Reconfiguration: The Stop-Sign Invariant

A configuration is a voter set, a pair of quorum sizes, and a configuration id. The application must see one sequence across configuration changes. A stop sign fixes which prefix belongs to the old configuration; the next configuration owns the application sequence after that boundary. `Replicated_Log_Node` wraps the core `Node` with entries that are either a command or a stop sign:

src/replicated_log.odin

```
// One log entry: an application command or a sealing stop sign.
Entry :: union(
    $Value: typeid,
    $MAX_MEMBERS: int = DEFAULT_MAX_MEMBERS,
    $MAX_METADATA_BYTES: int = DEFAULT_MAX_METADATA_BYTES,
) {
    Value,
    Stop_Sign(MAX_MEMBERS, MAX_METADATA_BYTES),
}
```

`Replicated_Log_Node` takes `Value`, `MAX_MEMBERS`, `WINDOW_SLOTS`, `CHUNK_SLOTS`, `MAX_METADATA_BYTES`, and `GATE`, and holds a `core` node over that union plus four seal fields: `configuration_id`, `stop_sign`, `stop_slot`, and `stop_pending`. Its effects type is `Effects(Entry(Value, MAX_MEMBERS, MAX_METADATA_BYTES), MAX_MEMBERS, WINDOW_SLOTS, CHUNK_SLOTS, GATE)`, and its durable state is the core's `Ledger(Entry(Value, MAX_MEMBERS, MAX_METADATA_BYTES), WINDOW_SLOTS)`, which `log_restore` takes and `replicated_log_ledger` (the `ledger` proc group) returns. A `Stop_Sign` carries the next `configuration_id`, the next member list, and up to `MAX_METADATA_BYTES` of opaque handover metadata, for example the identifier of a state image the new members must install.

Figure 14: A stop sign decided in slot s seals configuration C_1. Configuration C_2 continues the same slot line at $s+1$.

6.4.1 Proposing the stop

`log_reconfigure(&log, next_configuration_id, next_members, metadata, &effects)` is `replicated_log_propose_stop_sign` under a shorter name. It refuses an id that is not strictly greater than the current one with `.Configuration_Id_Regression`, validates the member list and metadata size through `stop_sign_create`, and proposes the stop sign through the ordinary `node_propose`. To the host it is one more entry in the pipeline; to the log it is the last one.

> **Definition: Sealed**
> `log_is_sealed(&log)` is true while a stop sign naming a newer configuration is pending or decided on this node. Pending means it sits in a used ledger cell, whether as an acceptor's vote or as the leader's own proposal. While sealed, `log_propose`, `log_propose_batch`, and a second `log_reconfigure` all return `.Log_Sealed` before touching the core.

The seal starts early on purpose. The proposer is sealed the moment `log_reconfigure` returns, because its proposal is its own vote in the ledger; an acceptor is sealed the moment it votes for the stop. If the stop is later overtaken (a higher ballot re-proposes that slot with a command, as recovery may) the pending flag is recomputed after every transition (`replicated_log_observe_effects`) and the seal clears. This local gate stops new proposals as soon as the node knows a stop is pending. Other owners may still decide later slots before learning the stop; the release rule below excludes those slots from the application log.

6.4.2 Observing the decision

Once the stop sign commits, `log_stop_sign(&log)` returns it with `true` and `log_stop_slot(&log)` returns its slot. After a crash, `log_restore` takes the replayed `Ledger` and rediscovers a committed stop by walking the ledger's

`chosen` bitmap; `log_pending_stop_sign(&log)` also reports an undecided one held in a used cell, so a host can resume its own handover phase without guessing.
The library decides where the boundary is. It does not move application state, start processes, or stop old traffic. The host waits for the decided stop, delivers every slot through it, transfers whatever the metadata names, and then starts the new configuration:

```
stop, sealed := paxos.log_stop_sign(&old)
if sealed {
    stop_slot := paxos.log_stop_slot(&old)
    anchor := paxos.Trim_Anchor{trim_id = 1, chosen_trim_slot = stop_slot}
    err := paxos.log_init_from_stop(&next, id, stop, stop_slot, anchor)
}
```

`log_init_from_stop(node, id, stop, stop_slot, anchor, options = {})` builds the membership from the stop sign's members, takes its configuration id, and calls `log_continue_at` with `stop_slot` as the floor: the new node's `delivered_through` and `memory_floor` are `stop_slot`, and its `next_slot` is `stop_slot + 1`. The slot line does not restart; the first command of C_2 takes slot $s + 1$. An id that is not among the stop sign's members is refused with `.Not_Member`, which is how a removed voter learns it is no longer one. An anchor above the floor is `.Trim_Regression`.

6.4.3 Fencing old traffic

A message from C_1 may still be in flight when C_2 starts, and a node that has not heard about the stop may keep sending. A bare `Envelope` carries no configuration, so the log offers `Log_Envelope`: a `configuration_id` next to the core `envelope`. On the way out, `log_envelope(&log, message)` stamps each outbound envelope with the sender's current configuration id. On the way in, `log_step` given a `Log_Envelope` is the checked overload (`replicated_log_step_checked`): it compares the stamp to the node's own id before the core sees the message, and on a mismatch resets the effects and returns `.Configuration_Mismatch` with no writes and no messages. The plain `Envelope` overload is for transports that already keep configurations apart.

```
for message in paxos.messages_slice(&effects) {
    deliver(paxos.log_envelope(&log, message))
}
// On the receiving node:
err := paxos.log_step(&log, stamped, &effects)
if err == .Configuration_Mismatch {
    // Stale traffic; nothing was written or sent. Drop it.
}
```

Do not relabel old traffic with the new id to make it pass. A vote cast under C_1's quorum rules is not a vote under C_2's. Under rotating ownership the member list also fixes which slots each member owns (`owner_of` deals the slot line in membership order), so a membership change is a stop sign there too. And because another owner may get a suggestion decided above the stop sign before it learns of the seal, the log abandons every decision above a decided stop sign: `replicated_log_observe_effects` cuts them from the released batch, the read procedures report them undecided, and the next configuration decides those slots afresh. The ownership chapter returns to this.

> **API anchor: `log_reconfigure`, `log_is_sealed`, `log_init_from_stop`, `log_envelope`, `log_step`**
> The reconfiguration surface of `Replicated_Log_Node`; every name is also available with its `replicated_log_` spelling. in `src/replicated_log.odin`

6.5 Catch-Up and Reconciliation

A node that missed traffic does not enter a special recovery mode. It asks a peer for decisions with `Learn_Message{from_slot, count}`, and `on_learn` answers by walking the ledger's `chosen` bitmap and sending a `Commit_Message` for every decision in that range that is still in its window. If any part of the range lies at or below the answering node's `memory_floor`, that part has left protocol memory, and the node emits a `Serve_Range_Request{peer, first, count}` in `effects.requests` for exactly that part: the host serves it from its own journal or state image, because the library never reads history it has released. A `from_slot` of zero, or a

`count` of zero or above `CHUNK_SLOTS`, is `.Invalid_Slot`, so one request never asks for more than one chunk. Three entry points send a `Learn_Message`:

1. `request_catch_up(&node, peer, from_slot, &effects)` asks `peer` for one chunk from `from_slot`. The host calls it when it knows it is behind, and again as its prefix advances.
2. `reconnected(&node, peer, &effects)` is the host's signal that a link came back. A leader answers by running `resend_to` for that peer; a follower whose `leader_hint` is that peer asks it for decisions from `delivered_through + 1`.
3. The core sends one itself when a heartbeat, an election manifest, or a resend sweep reveals a peer that has decided further than this node.

The last case relies on `peer_decided_through`, one slot per member, which `node_step` updates from every message that reports progress:

src/consensus.odin

```
// The sender's decided prefix, when the message kind reports it.
@(private)
message_decided_through :: #force_inline proc(message: Message($V)) -> (Slot, bool) {
    #partial switch msg in message {
    case Accepted_Message:      return msg.decided_through, true
    case Heartbeat_Message:     return msg.decided_through, true
    case Nack_Message:          return msg.decided_through, true
    case Promise_Range_Message: return msg.chosen_through, true
    case Prepare_Message:       return msg.first - 1, true
    case Learn_Message:         return msg.from_slot - 1, true
    }
    return 0, false
}
```

The leader uses it to skip resends a peer no longer needs and to notice when a peer is ahead of it. Decisions may arrive out of order; each is recorded in its cell and only the contiguous prefix is released by `emit_contiguous`.

6.6 Trim Anchors and State Images

The window bounds residency; the host's journal holds the rest. A journal cannot grow forever either, so the cluster periodically agrees that everything through some slot is captured in a state image, and acceptors then answer phase one for that prefix from the anchor instead of from cells:

src/ledger.odin

```
// The trim anchor: every slot at or below `chosen_trim_slot` is chosen and has been
// folded into a host state image identified by `trim_id`. The core compares anchors
// by identity; the host binds the image's checksum to the id itself.
Trim_Anchor :: struct {
    trim_id:          u64,
    chosen_trim_slot: Slot,
}
```

The anchor carries no hash. The core compares anchors by identity only: two anchors with the same `trim_id` must be the same anchor. A host that wants to verify a state image binds a monotonically increasing `trim_id` to the image's checksum in the durable record that chose the trim. A raw checksum is not a suitable sequence number: a later checksum can be numerically smaller and would fail the trim-regression check. How the anchor is chosen is the host's business; the natural way is to put the trim record in the log as an ordinary command, ordered with everything else. Once the host knows the record is chosen, it calls `install_chosen_trim(&node, anchor, &effects)`. The call refuses an anchor above `delivered_through` with `.Invalid_Slot`, and a lower `trim_id`, a different anchor under the same `trim_id`, or a lower `chosen_trim_slot` with `.Trim_Regression`; the same anchor again is `.None` with nothing written. Otherwise it emits `Write_Trim` for the journal, adopts the anchor into `ledger.anchor`, and raises `memory_floor` to the anchor slot, never past `delivered_through`.

From then on the anchor travels with the node. `on_prepare` skips cells at or below it and reports it in `Promise_Range_Message.anchor`; `on_accept` ignores votes for slots at or below it; and a candidate folds every reported anchor into the trim fence (`quorum_fences`), so no new leader can propose into a trimmed prefix.
A node so far behind that the slots it needs are below the cluster's anchor cannot catch up from decisions. Its host fetches the state image, verifies it against the anchor, persists it, and calls `begin_recovery(&node, anchor)`:

src/node.odin

```
// Installs a certified state image at `anchor`, keeping votes and decisions above it. The
// host must persist the image and the anchor before running further transitions.
node_begin_recovery :: proc(node: ^Node($V, $M, $W, $C, $G), anchor: Trim_Anchor) -> Error
{
    node_assert_valid(node)
    ledger_apply(&node.ledger, Write_Trim(anchor)) or_return
    node.leader_hint = nil
    node.election_ticks, node.heartbeat_ticks, node.resend_ticks = 0, 0, 0
    node.peer_decided_through = {}
    node.resend_cursor = {}
    node.role = .Follower
    clear_election(node)
    node_resume_at(node, anchor.chosen_trim_slot)
    return .None
}
```

`node_resume_at` clears every open vote at or below the anchor; every vote and decision above it is kept, because a vote above the anchor may already be part of a quorum that a future election must find. The node returns to `.Follower` with an empty election state and a floor at the anchor. `begin_recovery` emits no effects: the host must have persisted the image and the anchor before the call, since the node now answers phase one from them.
`continue_at(&node, id, membership, floor, anchor)` is the same idea for a node that starts empty rather than repairing in place: a fresh voter joining at a known floor, or the next configuration after a stop sign, which is what `log_init_from_stop` calls. An anchor above the floor is `.Trim_Regression` there too.

> **API anchor: `install_chosen_trim`, `begin_recovery`, `continue_at`, `trim_anchor`**
> Adopt a chosen anchor with a durable record; reset onto an installed image keeping votes above it; start empty at a floor; read the adopted anchor. in `src/node.odin`

6.7 Non-Voting Learners

Some processes need the decided sequence and nothing else: a read replica, an indexer, an auditor. Two shapes are available.
The first is a core `Node` initialized with `node_init_learner(&node, id, membership)`. Its `id` must be non-zero and lie outside the voting membership, otherwise `.Invalid_Node_Id` or `.Learner_Is_Voter`. It never promises, votes, or campaigns; `node_step` accepts only `Commit_Message` from a member and returns `.Learner_Message_Forbidden` for every other kind, and `tick` does nothing. A host that has certified a decision by some other path installs it with `learn_chosen(&node, from, slot, value, &effects)`, which is `.Not_Learner` on a voter and `.Not_Member` if `from` is not a member. The node releases the contiguous prefix through `effects.committed` exactly as a voter does; `Replicated_Log_Node` offers the same through `log_init_learner`, `log_restore_learner`, and `log_learn_chosen`.
The second is the standalone `Learner` in `src/learner.odin`, a window of `MAX_ENTRIES` cells with no membership at all:

src/learner.odin

```
// Result of recording a certified chosen value in the learner window.
Learn_Result :: enum {
```

```
    // Value was buffered in the window; a gap below it prevents immediate release.
    Buffered,
    // Contiguous released prefix advanced past this value.
    Advanced,
    // Duplicate of an already released or buffered value.
    Duplicate,
}
```

`learner_init(&l, configuration_id)` binds it to one configuration. `learner_learn_chosen(&l, configuration_id, slot, value)` returns a `Learn_Result` and an `Error`: `.Configuration_Mismatch` for a foreign configuration, `.Window_Full` when the slot is more than `MAX_ENTRIES` above `released_through`, and `.Conflicting_Chosen_Value` if the same slot arrives with a different value, which is a safety incident to preserve, not to retry. `learner_read_chosen(&l, from_slot, output)` copies the released suffix from `from_slot`, and `learner_chosen_at(&l, slot)` reads one released value. The learner takes and stores values by copy, not by pointer, because it has no ledger for a pointer to refer into. It trusts its caller: it has no quorum to check against, so the host must feed it only values it has certified as chosen.

6.8 Two Ways to Lead

Everything above assumes one elected leader: a node campaigns, wins a read quorum, and proposes in every slot above its term base until a higher ballot displaces it. `Node_Options.rotating_ownership` selects a second discipline in which the slot line is dealt round-robin and each member proposes in its own slots with no phase one at all, because round zero of each such slot's ballot space belongs to the owner alone. Under that option `campaign` returns `.Campaign_Disabled`, `proposal_gate` no longer requires the `.Leader` role, and `tick` runs `tick_ownership`: idle owners fill their slots with the no-op, a stalled prefix is repaired by a bounded phase one (`Prepare_Scope.Bounded`, recorded per decree as `Write_Promise_At`), and a suggestion that lost to such a revocation is proposed again. The ledger, the effect contract, the trim anchor, the learners, and reconfiguration are the same in both disciplines; `tools/check.py` runs the fault simulator in both. The next chapter derives the ownership rules from the same Synod proof and reads `src/ownership.odin` procedure by procedure.

Exercise 14.1
For five voters, choose read and write quorum sizes that make commits as cheap as possible while elections stay possible with two voters down. Show that the pair satisfies `Q1 + Q2 > N` and say what it costs.

Hint: Start from what "possible with two voters down" bounds, then let the inequality bound the other size.

Review & Discussion
Review advanced log maintenance and configuration mechanisms:

- Identify the conditions that trigger `Learn_Message` catch-up requests from followers versus leaders.
- Explain why `is_leader_caught_up` verifying prefix agreement is insufficient for local linearizable reads without clock or lease proofs.
- Formulate the stop-sign invariant for configuration transitions and name the error codes protecting sealed log epochs.
- Describe how trim anchors enable log compaction while safeguarding pending consensus slots.

7 Rotating Slot Ownership

Learning Objectives
After completing this chapter, you will be able to:
- Determine slot ownership deterministically without leader elections.
- Explain how reserving ballot round zero allows slot owners to propose without phase-one preparation while preserving invariant B1.
- Trace slot skip messages, revocation ballots, and suggestion resubmissions through the implementation.
- Evaluate the trade-offs of rotating ownership under idle versus crashed nodes.
- Identify the procedures in `src/ownership.odin`, `src/consensus.odin`, and `src/election.odin` that enforce rotating ownership safety.

7.1 One Leader Is a Bottleneck

The Multi-Paxos chapter made a single stable leader the engine of the log: it runs phase one once, and from then on every proposal is one round trip of `Accept` and `Accepted`. The leader coordinates each decision and can become a bottleneck. Every value a client hands to a follower must first cross the network to the leader, so a proposal from a non-leader pays a forwarding hop before its round trip, and the leader's outbound bandwidth, disk, and CPU bound the throughput of the whole group. In a group spread across sites the forwarding hop is a wide-area delay on every proposal that did not happen to originate at the leader.

Mao, Junqueira, and Marzullo's Mencius (OSDI 2008) removes the forwarding hop by dealing the log out round-robin: every member is the coordinator of its own instances, suggests values in them without a prepare, skips them when it has nothing to say, and is revoked when it is suspected. `paxos-odin` implements the same idea as an option on `Node`, called **rotating slot ownership**. Nothing in the acceptor's or learner's proof changes; what changes is who may propose where, and at which ballot.

API anchor: `Node_Options.rotating_ownership`
A `bool` in `Node_Options`. `node_init` copies it into `node.ownership` and sets `own_next` to the first slot this member owns. Every member of the group must be initialised with the same setting: a node without it ignores every round-zero `Accept_Message` in `on_accept`, and a node with it refuses `campaign` with `.Campaign_Disabled`. `Replicated_Log_Node` passes the option through `log_init`, `log_restore`, `log_continue_at`, and `log_init_from_stop`. in `src/node.odin`

7.2 The Ownership Rule

The membership is an ordered list: `membership_init` sorts the ids the host gives it, so every node holds the same list whatever order its host used, and a member's position in that list is its stable index. Ownership is a function of that index and the slot number alone:

src/ownership.odin

```
// The member that owns `slot`.
owner_of :: #force_inline proc(node: ^Node($V, $M, $W, $C, $G), slot: Slot) -> Node_Id {
    count := Slot(membership_count(&node.membership))
    return membership_get(&node.membership, int((slot - 1) % count))
}
```

Slot s is owned by the member at index $(s-1) \bmod N$. With members `{1, 2, 3}` in that order, member 1 owns slots 1, 4, 7, …, member 2 owns 2, 5, 8, …, and member 3 owns 3, 6, 9, …. The first test in `tests/test_ownership.odin` pins this down: `owner_of(&node, 1)` is 1 and `owner_of(&node, 5)` is 2. No message ever carries an owner id;

every member computes the same answer from the same membership. Hosts may pass the ids in different orders: `membership_init` sorts them into the same ascending order. They must still agree on the member set and quorum sizes for the configuration.

The private helper `own_slot_from(node, from)` returns the first slot at or after `from` that this node owns; `node_init` seeds `own_next` with `own_slot_from(node, 1)`, and `node_resume_at` recomputes it from `next_slot` after a restore so a restarted owner never re-suggests in a slot its ledger already holds.

> **API anchor: `owner_of`**
> `owner_of(node, slot) -> Node_Id`. Public and pure: a host that wants to route a client to the member that will suggest its value soonest can call it, but nothing requires that, because any member can propose at any time in its own slots. in `src/ownership.odin`

Slot	1	2	3	4	5	6
Owner	A	B	C	A	B	C
Value	X	hole	Z	-	-	-
Release	X ->	blocked	waits	-	-	-

B skips slot 2, or a read quorum recovers it at a higher ballot.
Only then can the application receive slots 2 and 3.

Figure 15: Ownership distributes proposals, but application order remains shared. A chosen value in slot 3 waits behind an empty slot 2. The owner can fill the hole with a no-op; if it is unavailable, recovery must first preserve any earlier vote.

7.3 Partitioning the Ballot Space

Phase one exists to make sure no lower ballot can still choose a different value in the same decree. An owner can skip it only if, in its own slots, there is no lower ballot at all. The library arranges exactly that by reserving one round of the 40-bit round field:

src/ballot.odin

```
//   bits 63..24  round      (40 bits, the campaign counter; round 0 is reserved for
//                            slot owners under rotating ownership)
```

`start_campaign` and `start_revocation` both compute `greatest + 1` for a round, so every campaign ballot has round one or more. The owner's ballot is round zero with priority zero:

src/ownership.odin

```
// The ballot an owner proposes with: round zero, which no campaign ever uses.
ownership_ballot :: #force_inline proc(owner: Node_Id) -> Ballot {
    return ballot_make(0, 0, owner)
}
```

The acceptor closes the partition. Before anything else in `on_accept`, a round-zero accept is checked against the ownership rule:

src/consensus.odin

```
    // Round zero belongs to the slot's owner alone (B1 per decree under rotating
ownership).
    if ballot_round(msg.ballot) == 0 {
        if !node.ownership || ballot_node(msg.ballot) != owner_of(node, msg.slot) do
return .None
    }
```

So in decree s the only round-zero ballot any acceptor will ever vote at is `ownership_ballot(owner_of(s))`. Lamport's B1 (Section 2.2 of *The Part-Time Parliament*) asks that each ballot in a decree have a unique number,

and it is stated per decree: two decrees may reuse the same number freely. Under ownership the ballot `(0, 0, 3)` appears in every slot member 3 owns, and in each of those decrees it is unique and the least ballot anyone can vote at. B3 then lets the owner pick any value, because the set of votes at lower ballots is empty by construction. Campaign and revocation ballots at round one and above are compared exactly as before, so everything the single-decree chapter proved holds unchanged.

7.4 Proposing Without Phase One

With ownership on, `proposal_gate` returns `.None` for any voting member regardless of role, and `node_propose` hands the value to `propose_owned`:

src/ownership.odin

```
// Proposes `value` in this node's next usable own slot.
@(private)
propose_owned :: proc(
    node: ^Node($V, $M, $W, $C, $G),
    value: V,
    effects: ^Effects(V, M, W, C, G),
) -> (slot: Slot, err: Error) {
    for {
        slot = next_usable_own_slot(node) or_return
        err = send_accept(node, slot, ownership_ballot(node.id), value, effects)
        if err != .Not_Leader do break
        // A revoker's promise reached this slot first; the next own slot is ours.
        node.own_next = own_slot_from(node, slot + 1)
    }
    err or_return
    node.own_next = own_slot_from(node, slot + 1)
    node.highest_seen = max(node.highest_seen, slot)
    return slot, .None
}
```

`send_accept` is the same phase-two procedure a leader uses, now with the ballot as a parameter: it claims the cell, records the node's own vote and `Write_Vote`, registers the slot in `lead_slot` and `lead_ballot` so that `on_accepted` will count acknowledgements for exactly this ballot, and broadcasts the `Accept_Message`. A write quorum of one commits on the spot, as before.

`next_usable_own_slot` decides which slot that is: it asks `own_slot_probe`, which starts from `own_next` and steps over any own slot that is already decided or whose per-decree promise is above the owner's ballot, that is, a slot a revocation has fenced, without changing any state; only then does `own_next` move:

src/ownership.odin

```
        cell := cell_of(slot, W)
        occupant := l.slot[cell]
        switch {
        case occupant == slot:
            if l.state[cell] != .Chosen && ledger_promise_for(l, cell) <= mine do
return slot, .None
        case occupant == 0 || (occupant <= node.memory_floor && l.state[cell] == .Chosen):
            if l.promised <= mine do return slot, .None
        case:
            // The cell still holds an older open slot; wait for the host to release it.
            return 0, .Window_Full
        }
        slot = own_slot_from(node, slot + 1)
```

Two errors come out of it. `.Global_Slot_Exhausted` when `own_next` reaches `max(Slot)`, and `.Window_Full` when the candidate slot lies more than `WINDOW_SLOTS` above `memory_floor`. The second is the backpressure of a

leaderless log: if some slot below is not decided and the host has not consumed the prefix, the floor does not move, and every owner eventually gets `.Window_Full` until a revocation fills the hole. `node_propose_batch` first runs `own_slots_available`, which probes the `len(values)` own slots the batch would take, stepping over revoked and decided ones exactly as a proposal would, without changing anything; only when every one of them fits under the window edge does it call `propose_owned` once per value. So a batch is admitted whole or refused whole, and a refused batch leaves no vote behind. A batch of k values spans at least $(k-1)N+1$ slots of the log, and the slots in between belong to the other owners.

> **API anchor: `propose`, `propose_batch`**
> Unchanged signatures. Under ownership `propose` returns the own slot it suggested in, never `.Not_Leader` or `.Leader_Catching_Up`, and the batch form fills consecutive own slots. in `src/consensus.odin`

7.5 Skips

A log with three owners and one busy member would stall after the busy member's first slot: slot 2 belongs to member 2, and until it is decided nothing above it can be released. Mencius solves this with skips. Here a skip is an ordinary suggestion of the host's no-op, sent by `tick`:

src/ownership.odin

```
// At most this many skips leave per tick, so an idle owner catching up does not flood.
SKIP_BURST :: 8

// Skips: no-ops in this node's own slots below the highest slot anyone has reached, at
// most `budget` (and never more than SKIP_BURST) per tick.
@(private)
skip_idle_slots :: proc(
    node: ^Node($V, $M, $W, $C, $G),
    noop: V,
    budget: int,
    effects: ^Effects(V, M, W, C, G),
) -> (sent: int, err: Error) {
    for node.own_next <= node.highest_seen && sent < min(budget, SKIP_BURST) {
        _, propose_err := propose_owned(node, noop, effects)
        if propose_err == .Window_Full do break
        if propose_err != .None do return sent, propose_err
        sent += 1
    }
    return sent, .None
}
```

`highest_seen` is the greatest slot this node has any evidence of: its own suggestions raise it in `propose_owned`, an accept for a slot raises it in `on_accept` (after the ownership check), and a decision raises it in `record_commit`. A restore sets it from the ledger in `node_resume_at`. The rule is simply "if somebody has reached a slot above one of mine, my slot is holding them up, so fill it." Each tick sends at most `min(CHUNK_SLOTS, SKIP_BURST)` skips so an owner that wakes up far behind does not flood the network or its own effects buffer.

Consider the second test, `ownership_idle_owners_skip`. Members `{1, 2, 3}`, only member 1 has traffic:

1. Member 1 proposes 11. `own_next` is 1, so the suggestion goes to slot 1 at ballot `(0, 0, 1)`; `own_next` becomes 4 and `highest_seen` 1.
2. Member 1 proposes 14. Slot 4, `own_next` 7, `highest_seen` 4.
3. Members 2 and 3 process the two accepts. Both are round zero from the right owner, both are voted, and both raise `highest_seen` to 4. Slots 1 and 4 are decided everywhere, but `delivered_through` stays at 1 because slot 2 is empty.
4. On the next tick, member 2 finds `own_next = 2 <= 4` and suggests the no-op in slot 2; then `own_next` is 5, above `highest_seen`, and the loop stops. Member 3 does the same for slot 3.

5. The skips are decided like any other value, by a write quorum of votes at round zero. Once they are, every member releases slots 2, 3, and 4 and `decided_through` is 4 on all three.

A skip is not free: it costs the same messages and the same durable votes as a value. What it buys is that nobody has to guess whether an idle owner is slow or dead, because a live idle owner fills its slots within a tick.

7.6 Retransmission and Catch-Up Without a Leader

`node_tick` hands the whole tick to `tick_ownership` when ownership is on. There are no heartbeats, because there is no leader to announce. Instead every member does what a leader used to do for the slots it drives: every `resend_interval_ticks` it calls `resend_to` for each peer. `resend_to` resends a `Commit_Message` for every decided cell above what the peer has reported decided, and an `Accept_Message` for every voted cell where `lead_slot` names the slot and the vote's ballot equals `lead_ballot`. That key is what makes retransmission correct in a log with many proposers: an acceptor that merely voted for someone else's suggestion has no `lead_ballot` entry for it and does not resend it, while an owner resends its own suggestions at `ownership_ballot(node.id)` and a revoker resends its accepts at its campaign ballot. `on_accepted` uses the same key, so a late acknowledgement for a ballot the node is no longer driving in that slot is dropped.

Catch-up follows the ownership rule. When the prefix is stalled, every `heartbeat_interval_ticks` of stall `tick_ownership` sends a `Learn_Message` to `owner_of(delivered_through + 1)`, the one member that certainly knows what became of the stuck slot if it is alive. `peer_decided_through`, updated from every `Accepted_Message`, still lets `resend_to` ask a peer that has decided further for what it has.

> **Prediction Exercise**
> Members `{1, 2, 3}` have decided slots 1, 2, and 4, and member 3, the owner of slot 3, has crashed without ever suggesting anything there. Which member notices, after how many ticks, and what message does it send first? When it finally runs phase one, which slots does its `Prepare_Message` name, and why not the whole chunk?

7.7 Revocation

A crashed owner leaves a hole that no skip can fill. The library repairs it by running phase one, but a phase one narrowed in two ways: it covers only the stalled range, and it promises per decree, so the crashed owner is fenced out of those slots only and keeps every other slot it owns.

Stall detection. `tick_ownership` compares `delivered_through` with `highest_seen`. Equal or above means nothing is outstanding and `stall_ticks` resets to zero. Otherwise `stall_ticks` increments, and when it reaches `election_timeout_ticks` (the same option a follower uses to suspect a leader) the node calls `start_revocation`. This answers the first half of the prediction: both survivors notice, on the tick where `stall_ticks` reaches the timeout, and before that each has sent a `Learn_Message` to member 3 every `heartbeat_interval_ticks`.

Bounded prepare. `start_revocation` picks a fresh round above everything it has seen, exactly as `start_campaign` does, bounds the range, promises itself, and only then enters `.Preparing`, remembers the no-op, and clears the election state:

src/ownership.odin

```
    base := node.delivered_through + 1
    chunk_end := slot_add(base, Slot(C - 1))
    last := min(chunk_end, max(node.highest_seen, base), slot_add(node.memory_floor,
Slot(W)))
    prepare := Prepare_Message{
        ballot = ballot_make(greatest + 1, node.priority, node.id),
        first = base, last = last, scope = .Bounded,
    }
    // The revoker promises itself first, so its ballot is durable before the Prepare
    // leaves and a restart campaigns above it (the same rule as start_campaign). If a
    // cell of the range is still held by an older open slot, nothing has changed yet:
    // the node stays a follower and tries again after the next timeout.
```

```
    if !promise_bounded(node, prepare, effects) {
        node.stall_ticks = 0
        return .None
    }
    node.ballot = prepare.ballot
    node.role = .Preparing
```

The revoker answers its own prepare before sending it: `promise_bounded` records a `Write_Promise_At` for every slot of the range in the same batch as the `Prepare`, so the revocation ballot is durable before any peer hears of it. That is what lets a restarted revoker always pick a higher round, and the safety-argument chapter's pre-durable lemma rests on it.

The range is $[\texttt{delivered\_through + 1}, \min(\texttt{chunk end}, \texttt{highest\_seen}, \texttt{memory\_floor + W})]$, clamped to the live window so the revoker can always promise every slot of it. There is no reason to fence any slot above `highest_seen`: nobody has reached it, its owner has not been slow about it, and revoking it would only steal a slot from a live member. That is the second half of the prediction. `Prepare_Message.scope` is `.Bounded`; a campaign's prepare leaves it at the zero value `.Global`.

Per-decree promises. `on_prepare` dispatches on the scope. A global prepare raises the ledger's single `promised` ballot and writes `Write_Promise`. A bounded prepare goes through `promise_bounded`, which promises each decree in the range separately in `promised_at[cell]` and writes one `Write_Promise_At` per slot. The effective promise for a cell is `ledger_promise_for`, the larger of the two, and both `on_accept` and `next_usable_own_slot` consult it. The procedure fails closed:

src/election.odin

```
    if msg.last - msg.first >= Slot(C) do return false
    for slot := msg.first; slot <= msg.last; slot += 1 {
        if slot <= node.memory_floor do continue
        cell, ok := claim_live(node, slot)
        if !ok || msg.ballot < l.promised_at[cell] do return false
    }
```

If the range is wider than one chunk, if a slot in it has no cell to carry its promise, or if any slot is already promised to a higher ballot, the acceptor sends nothing at all, not even a partial answer. A candidate can then never count a promise that was not recorded, and it simply times out and tries again at a higher round. A bounded prepare does not call `observe_leader`: the revoker is nobody's leader, and it must not reset anyone's timers. Any round-zero accept that arrives after the promise is refused with a `Nack_Message` whose `slot` names the fenced decree.

Resolution and B3. Promises flow back through the same `on_promise` and `on_promise_range` handlers a campaign uses. When a read quorum has described the range, `resolve_chunk` drives it. Under ownership `drive_all` is true, so every slot in the range is decided: a known decision is rebroadcast, a recovered vote is re-proposed at the revocation ballot (B3: the crashed owner's value survives if any quorum member voted for it), and an empty decree gets the no-op. Driving every promised slot matters here more than in a campaign, because a promised slot that was left undecided would fence its owner out of it forever. If an acceptor reported used cells above the range, `begin_next_chunk` continues with another bounded prepare, again capped at `highest_seen`.

No standing leader. When the range is driven, `become_leader` takes the ownership branch: role back to `.Follower`, `stall_ticks` to zero, `emit_contiguous` to release whatever became contiguous. There is no term, no `leader_base`, and nothing for `is_leader_caught_up` to say. Should a revoker stall again, `tick_ownership` starts a fresh revocation at a higher round after `election_timeout_ticks` in `.Preparing`.

Here is the third test, `ownership_revokes_a_crashed_owner`, with `Node(u64, 3, 16, 4)`, default timers, member 3 silent, and member 1 reaching the timeout first:

Step	Actor	Event and reason
1	M1, M2	`propose`: 11 in slot 1 by member 1, 12 in slot 2 by member 2, 14 in slot 4 by member 1, all at round zero. Every accept to member 3 is lost. Slots 1, 2, and 4 are decided by the votes of members 1 and 2; `delivered_through` is 2 and `highest_seen` is 4 on both.
2	M1, M2	`tick_ownership`: no resubmits, no skips (`own_next` is 7 and 5, both above 4). `delivered_through < highest_seen`, so `stall_ticks` becomes 1. At 3, 6, and 9 each sends `Learn_Message{from_slot = 3}` to `owner_of(3) = 3`, which is lost.
3	M1	`stall_ticks` reaches `election_timeout_ticks = 10`. `start_revocation`: ballot `(1, 0, 1)`, role `.Preparing`, `recover_base = 3`, chunk end 6, `recover_last = min(6, 4) = 4`. Sends `Prepare{(1,0,1), first = 3, last = 4, scope = .Bounded}` to 1, 2, and 3.
4	M1	`on_prepare` on its own prepare: `promise_bounded` claims a cell for slot 3 and finds slot 4's cell. Writes `Write_Promise_At (1,0,1)` for 3 and for 4. Reports slot 4 as `.Chosen` with 14, then `Promise_Range{reported = 1, more = false, chosen_through = 2}`.
5	M2	`on_prepare`: the same two `Write_Promise_At` records, the same report for slot 4, the same manifest. Its `highest_observed_round` becomes 1; its `leader_hint` and `election_ticks` are untouched.
6	M1	`maybe_resolve_chunk`: two complete descriptions reach the read quorum. `resolve_chunk`: the chosen fence is 2, so the drive starts at 3. Slot 3 has no recovered vote: `send_accept(3, (1,0,1), noop)`, writes `Write_Vote`, sends `Accept` to 2 and 3. Slot 4 is chosen locally: rebroadcasts `Commit{4, 14}`. `become_leader`: role `.Follower`, `stall_ticks = 0`.
7	M2	`on_accept` for slot 3 at round one: the owner rule does not apply, `(1,0,1)` is not below `promised_at`, so it votes, writes `Write_Vote`, calls `observe_leader` (hint 1), and sends `Accepted{(1,0,1), 3}`.
8	M1	`on_accepted`: `lead_slot[cell] == 3` and `lead_ballot[cell] == (1,0,1)`; two acknowledgements reach the write quorum. `record_commit(3, 0)` writes `Write_Chosen` and `emit_contiguous` releases slots 3 and 4. Broadcasts `Commit{3, 0}`.
9	M2	`on_commit`: records slot 3, releases 3 and 4. On both survivors `decided_through` is now 4, `stall_ticks` is 0, and `role` is `.Follower`, which is what the test asserts.

If member 2 reaches the timeout in the same round, it starts its own revocation at `(1, 0, 2)`, which is the higher ballot. Member 1 gets no answer to its prepare from member 2 (`promise_bounded` fails closed against the higher per-decree promise), member 2 resolves the range instead, and member 1 drops back to `.Follower` when it votes for member 2's round-one accept in `observe_leader`. Either way slot 3 takes the no-op and neither survivor keeps a role.

The fourth test, `ownership_revocation_keeps_a_seen_vote`, is the B3 case: member 3 suggests 33 in slot 3 and only member 2 hears it before member 3 falls silent. Member 2's `Promise_Message` for slot 3 reports a `.Voted` cell at `(0, 0, 3)` with 33, `resolve_chunk` re-proposes 33 at the revocation ballot, and slot 3 decides 33, not the no-op.

7.8 Resubmission

A revocation can decide the no-op in a slot whose owner had suggested a real value that nobody heard. The value was accepted from a client and must not vanish. Two places notice the case. `on_accept`, when the revoker's accept is about to overwrite the owner's own vote, and `record_commit`, when the decision arrives without the accept having been seen:

src/consensus.odin

```
    // An owner whose suggestion lost to a revocation proposes it again later.
    if node.ownership && l.state[cell] == .Voted &&
       l.vote_ballot[cell] == ownership_ballot(node.id) && l.value[cell] != value {
        queue_resubmit(node, l.value[cell])
    }
```

The conditions read: this node's own vote in the slot is at its ownership ballot, and the decision that just arrived carries a different value. The lost value goes into `resubmit`, a `small_array` of `CHUNK_SLOTS` values, and the next `tick_ownership` calls `drain_resubmits`, which `propose_owneds` each queued value in order into the next usable own slot, stopping (without losing anything) on `.Window_Full`. The fifth test, `ownership_revoked_suggestion_is_resubmitted`, silences member 3 after its suggestion of 33 for slot 3 reaches nobody, lets members 1 and 2 revoke slot 3 to the no-op, then reconnects member 3. The retransmitted `Commit{3, 0}` fires the hook, and 33 is decided in a later slot owned by member 3.

What a host may rely on is narrower than "never lost." The queue holds one chunk of values; `queue_resubmit` counts a value beyond that in `resubmits_dropped` (read it with `paxos.resubmits_dropped`) and leaves it to the host's ordinary timeout-and-retry discipline, the same one it needs anyway for a suggestion whose owner crashes with a non-durable vote. Resubmission is a liveness courtesy that covers the common case, not a delivery guarantee, and a resubmitted value is decided in a different slot from the one it was first suggested in, later in the log.

7.9 Why Round-Zero Accepts Wait for the Barrier

The bounded-core chapter introduced `pre_durable_messages`: an `Accept_Message` at a campaign ballot may leave before the sender's own writes are durable, because it asks peers to persist their votes and claims nothing about the sender, and a restarted proposer always campaigns at a fresh ballot. An owner's suggestion is different:

src/effects.odin

```
// ... An owner's round-zero suggestion is the
// exception: its own vote is the only durable record that the instance was used, so it
// must wait for the barrier or a restart could reuse the ballot for another value.
```

`pre_durable_next` therefore yields only accepts with `ballot_round > 0`. The failure it prevents is concrete. An owner suggests `a` in slot 5 at `(0, 0, 2)`, the accept leaves, and the owner crashes before `Write_Vote` reaches disk. It restarts, `node_resume_at` finds slot 5 empty, `own_next` is 5 again, and the next proposal `b` goes out in slot 5 at the very same ballot `(0, 0, 2)`. Two values under one ballot in one decree is the violation of B1 that everything else rests on, and it is exactly what the simulator's vote-level oracle reports: `persist_sim_write` keeps every durable vote per member and slot and fails the run with "ballot accepted two values in slot" the moment two members hold different values under the same ballot. That oracle is the reason round zero is excluded from the pre-durable path.

> **Warning: Round-zero accepts are not pre-durable**
> A host that sends accepts early through `pre_durable_messages` must send an owner's suggestions only after `confirm_writes_durable`. The iterator enforces this by never yielding a round-zero accept; a host that bypasses the iterator and drains `messages_slice` before the barrier under the `.Host_Managed` gate takes on the ballot-reuse hazard itself.

7.10 What Ownership Does Not Do

- **No leases and no read path.** Ownership is about who may propose. It says nothing about who may answer a read from local state; that still needs a quorum round trip or a barrier through the log, and leader leases remain a proposal (POD 0004). `current_leader` under ownership reports only the sender of the last commit or revocation accept this node processed.
- **No separate ownership order.** Membership order is ownership order. Changing the order changes every owner, so it changes only with the membership, at a stop sign, where the new configuration recomputes `owner_of` from its own member list and `Log_Envelope` keeps the old configuration's suggestions out. One rule makes that safe: an

owner that has not yet heard of the stop sign can still get a suggestion decided above it, so `Replicated_Log_Node` abandons every decision above a decided stop sign. It is never released and never readable, and the next configuration decides that slot again with its own quorums. The client whose command was abandoned sees a timeout and retries, as after any other loss.

- **No unbounded buffering.** A full-window stall applies backpressure. If a revocation cannot complete (say, no read quorum), owners receive `.Window_Full` from `propose` once their next own slot is `WINDOW_SLOTS` above the floor, and the host waits, exactly as it would behind a leader with a full window.
- **No campaigns.** `campaign` returns `.Campaign_Disabled`. The only phase one that runs is a revocation, started by a tick.

7.11 Evidence

`tests/test_ownership.odin` runs five scenarios on `Node(u64, 3, 16, 4)`, with an in-process queue and a `silent` member whose traffic is dropped:

- `ownership_three_owners_propose_concurrently`: four proposals from three owners decide in slots 1 through 4 without a campaign; `campaign` is refused.
- `ownership_idle_owners_skip`: slots 2 and 3 decide the no-op.
- `ownership_revokes_a_crashed_owner`: slot 3 is revoked to the no-op and both survivors end as `.Follower`.
- `ownership_revocation_keeps_a_seen_vote`: the revoker re-proposes 33 (B3).
- `ownership_revoked_suggestion_is_resubmitted`: 33 is decided in a later own slot once its owner returns.

The simulator (`sim/main.odin`) takes `--ownership`, which initialises every node with `rotating_ownership = true`, skips the bootstrap campaign, and lets `propose_random` pick any live node as the proposer. Every oracle from the single-leader runs applies unchanged: agreement and validity in `record_decision`, the durable-vote quorum and one-value-per-ballot checks in `persist_sim_write`, promise regression for the global promise, contiguity of released entries, convergence of every node onto the golden log, and the liveness probe after healing, which in ownership mode is proposed by whichever node the loop reaches first. Crashes still land at any of the three points of the host commit sequence, and a restarted node comes back through `restore` with the same option, which is how `node_resume_at`'s recomputation of `own_next` and `highest_seen` is exercised.

7.12 Single Leader or Rotating Ownership?

Property	Single stable leader	Rotating ownership
Round trips per proposal	One accept round trip at the leader; a proposal that originates elsewhere first crosses to the leader.	One accept round trip at the owner; no forwarding.
Messages per decision, N members	$N-1$ accepts, up to $N-1$ acknowledgements, $N-1$ commits, plus periodic heartbeats.	The same per slot, no heartbeats; but every idle owner's skip is one more decision.
An idle member	Costs nothing.	Fills its own slots below `highest_seen` with the no-op, at most `SKIP_BURST` per tick.
A crashed member	A crashed follower costs nothing while a quorum remains; a crashed leader costs an election after `election_timeout_ticks` with a global prepare.	Its slots below `highest_seen` stall the prefix for `election_timeout_ticks`, then a bounded revocation decides them; it keeps every slot above.
Campaign availability	`campaign` unless `campaign_disabled`.	`.Campaign_Disabled`, always.

Property	Single stable leader	Rotating ownership
Standing role after repair	A leader with a term base.	None; the revoker returns to `.Follower`.

Choose ownership when proposals originate at many members and forwarding latency dominates, and when members are rarely idle or rarely crash for long. Choose a single leader when most traffic already arrives at one place, or when an idle member must cost nothing.

Exercise 15.1
Three owners `{1, 2, 3}`. Member 2 crashes after proposing v in slot 5 with its accept delivered to member 3 only. Trace the revocation by member 1 and say which value slot 5 takes, and why member 2 will or will not resubmit.

Hint: Start with `owner_of(5)` and with what member 1's `highest_seen` must be for it to stall on slot 5 at all. Then ask what member 3's `Promise_Message` for slot 5 reports, what `resolve_chunk` does with a recovered `.Voted` cell, and which of the four conditions in `record_commit`'s resubmit hook fails when member 2 finally processes the commit for slot 5.

Review & Discussion
Analyze the rotating slot ownership protocol mechanics:

- Why a designated slot owner may propose at round zero without a phase-one read quorum, whereas an overriding candidate must run phase one.
- How idle nodes prevent head-of-line blocking by issuing skip entries.
- The mechanism by which stalled slots are revoked at higher ballot rounds.
- The four conditions required to trigger automatic resubmission of revoked suggestions.

8 Writing Reviewable Consensus Code

Learning Objectives
After completing this chapter, you will be able to:

- Identify the core Odin language patterns used throughout the consensus engine.
- Understand the rationale behind explicit context passing, compile-time assertions, and contiguous memory layouts.
- Distinguish between expected runtime `Error` returns, compile-time assertions, and unrecoverable process halts.
- Conduct thorough code reviews for consensus modifications structured like mathematical proofs.

8.1 The Zen of Odin for InsanAI

Before any rule, the creed. Every guideline in this chapter is a consequence of one of these lines, and every review comment on this repository can cite one.

> *Data is real; code is just the stream.*
> *Explicit is better than a hidden scheme.*
> *Simple blocks beat abstractions built too high,*
> *A mere mortal should see how the segments tie.*
> *Keep close to the metal, let allocations show,*
> *Pass your contexts cleanly so the lifetimes flow.*
> *Errors are values, never cast aside,*
> *Handle them explicitly; let nothing hide.*
> *Fail with grace, let diagnostics guide:*
> *Show the break, the hint, the fix inside.*
> *Design for speed, let safety lead the pace,*
> *Waste no cycle, leave no leaking trace.*
> *Keep it simple to use, explain, and maintain,*
> *So years from now, the logic remains plain.*
> *Coherence beats purity when real problems strike,*
> *But structure your memory as hardware would like.*
>
> - The Zen of Odin for InsanAI, POD 0001

The creed has teeth. `tools/check_style.py` enforces the structural constraints on every Odin file, and `make vet`, `make check`, and `paxodin check` run it first:

Constraint	Limit	How this repository meets it
File boundary	At most 1,408 physical lines per file.	The core is ten files, each one concern: `ballot.odin`, `bit_set.odin`, `membership.odin`, `ledger.odin`, `messages.odin`, `effects.odin`, `node.odin`, `election.odin`, `consensus.odin`, `ownership.odin`; around them `replicated_log.odin`, `learner.odin`, `errors.odin`, and `paxos.odin`.

Constraint	Limit	How this repository meets it
Line width	99 columns soft, 108 columns hard (tabs count as four); a longer line fails the build.	Long parametric signatures are broken one parameter per line.
Procedure density	At most 70 lines of logic per body; blank, comment, and divider lines do not count.	`EXPLANATIONS` is a data table, `node_step` is a dispatch, phase one is `on_prepare`, `on_promise`, `on_promise_range`, `maybe_resolve_chunk`, and `resolve_chunk` rather than one procedure.
Elm-style diagnostics	Context, hint, and remediation in every error.	`Error` plus `explain_error`; the durability gate's banner; every `#assert` message.
Performance and longevity	Mechanical sympathy, safety by visibility, the mere-mortal explainability test.	Columns and bitmaps in the `Ledger`, a ballot in one integer, inline buffers; `where` clauses and the runtime gate; no clever code that a reviewer cannot restate.

8.2 Why Zero Allocation Matters

A transition that can fail on allocation has one more failure path than the proof accounts for. Allocation failure is not itself a safety violation; a node that stops cleanly on it stays fail-stop. But it adds latency variance, fragmentation, and ownership questions to every transition. The library removes the question instead of answering it.

No procedure in `src/` calls `context.allocator`, `new`, or `make`. Every capacity is a type parameter: `MAX_MEMBERS`, `WINDOW_SLOTS`, and `CHUNK_SLOTS` on `Node` and `Effects`, `WINDOW` on `Ledger`, `MAX_METADATA_BYTES` on `Replicated_Log_Node`, and `MAX_ENTRIES` on `Learner`. The size of a node, a batch, and a ledger is therefore a compile-time constant, and the exact per-transition maxima in the `Effects` comment are the reason a batch can never overflow. Values are not copied into batches either: a write, a message, or a committed entry points into the ledger, and the host's transport and journal determine how many copies occur. The hosts that surround the core are free to allocate: the counter uses `core:container/queue` for its network and `tests/harness.odin` keeps its journal and its queue in a `[dynamic]` array. The line is drawn at the package boundary, not at the process.

8.3 The Odin Idioms This Library Uses

Each idiom below appears in the source as quoted. They are not decoration; each one removes a class of review question.

8.3.1 Struct of arrays, with bitmaps for the scans

The ledger is not an array of cell structs. Each of Lamport's variables is its own column, and two bitmaps say which cells are worth visiting:

src/ledger.odin

```
    slot:        [WINDOW]Slot,
    promised_at: [WINDOW]Ballot,
    vote_ballot: [WINDOW]Ballot,
    state:       [WINDOW]Cell_State,
    value:       [WINDOW]Value,
    // Bitmaps over cells: `used` has a vote or a decision, `chosen` has a decision.
    used:        Bit_Set(WINDOW),
    chosen:      Bit_Set(WINDOW),
```

A phase-one answer walks `used` and reads `slot`, `vote_ballot`, and `state`; the metadata walk does not load the payload bytes. Producing and consuming the reply still costs work: the host copies or serialises each reported value. The walk itself is `bit_set_next`, which counts trailing zeros in a 64-bit word instead of testing every cell:

src/ledger.odin

```
// The greatest slot held by any used cell.
ledger_highest_used :: proc(l: ^Ledger($Value, $WINDOW)) -> Slot {
    highest: Slot
    cell, more := bit_set_next(l.used, 0)
    for more {
        highest = max(highest, l.slot[cell])
        cell, more = bit_set_next(l.used, cell + 1)
    }
    return highest
}
```

The same loop shape answers a `Learn_Message` over `chosen` and drives a resend sweep over `used`. The reviewer's question changes from "did this loop visit every cell?" to "is the bitmap kept in step with the `state` column?", and `node_assert_valid` asks exactly that.

8.3.2 A ballot is one integer

Lamport's B1 needs a total order on ballots. The library packs the round, the priority, and the proposer into one `u64` so that the order is integer comparison and every record and message carries eight bytes:

src/ballot.odin

```
Ballot :: distinct u64

BALLOT_ZERO      :: Ballot(0)
BALLOT_ROUND_BITS :: 40
MAX_ROUND        :: u64(1) << BALLOT_ROUND_BITS - 1

ballot_make :: #force_inline proc(round: u64, priority: u8, node: Node_Id) -> Ballot {
    return Ballot(round << 24 | u64(priority) << 16 | u64(node))
}

ballot_round :: #force_inline proc(b: Ballot) -> u64 {
    return u64(b) >> 24
}
```

`distinct` matters: a `Ballot` is not a `u64` to the compiler, so a slot or a count cannot be passed where a ballot belongs, while `<` and `max` still work because the underlying type is an integer. The fields are read with `ballot_round`, `ballot_priority`, and `ballot_node`, never with a shift at a call site.

8.3.3 `#force_inline` helpers

The one-line accessors on the hot path are marked `#force_inline`, so a helper that exists for naming costs no call:

src/ballot.odin

```
// The window index of a slot. WINDOW is a power of two, so this is one mask.
cell_of :: #force_inline proc(slot: Slot, $WINDOW: int) -> int {
    return int((slot - 1) & Slot(WINDOW - 1))
}
```

`ledger_promise_for`, `ledger_cell`, `ledger_record_vote`, the `bit_set_` accessors, and the `effects_add_` producers follow the same rule. The attribute is reserved for bodies a reviewer can read in one glance; a procedure with a loop or a branch that matters is a plain `proc`.

8.3.4 Pointers into the ledger instead of copies

A transition never copies a value into its batch. The `Committed` entry, the `Write_Vote`, and the `Accept_Message` all point at the ledger cell:

src/consensus.odin

```
    l.promised_at[cell] = max(l.promised_at[cell], ballot)
    ledger_record_vote(l, cell, ballot, value)
    effects_add_write(effects, Write_Vote(V){ballot = ballot, slot = slot, value =
&l.value[cell]})
```

The lifetime is the contract stated in `src/messages.odin`: the pointer is valid until that node's next transition, and every transition begins with `effects_reset`. This is "pass your contexts cleanly so the lifetimes flow" in its narrowest form. The host's side of the idiom is the `Packet` in `examples/counter.odin`, which copies the value at enqueue time.

8.3.5 Proc groups

Odin's procedure groups let one verb serve several receiver types. The public surface in `src/paxos.odin` is a list of such groups:

src/paxos.odin

```
campaign             :: proc{node_campaign, replicated_log_campaign}
propose              :: proc{node_propose, replicated_log_propose}
propose_batch        :: proc{node_propose_batch, replicated_log_propose_batch}
step                 :: proc{node_step, replicated_log_step, replicated_log_step_checked}
```

`paxos.propose(&node, value, &effects)` resolves to `node_propose` when the first argument is a `^Node` and to `replicated_log_propose` when it is a `^Replicated_Log_Node`; `paxos.step` further distinguishes a bare `Envelope` from a `Log_Envelope` that carries a configuration id. `paxos.init` covers six receivers, from `Node` to `Stop_Sign`. The long spellings stay available for a call site that wants to name its receiver.

8.3.6 Parametric structs with defaults and `where` clauses

The `Node` header from the library chapter shows all three features at once: `$Value: typeid` makes the struct generic over the command type, `$MAX_MEMBERS:`
`int = DEFAULT_MAX_MEMBERS` gives each capacity a default, and `where intrinsics.type_is_comparable(Value)` rejects a value type the library could not compare. A procedure over such a struct binds the parameters once with `$` and then spells the batch with the bound names:

src/consensus.odin

```
node_propose :: proc(
    node: ^Node($V, $M, $W, $C, $G),
    value: V,
    effects: ^Effects(V, M, W, C, G),
) -> (slot: Slot, err: Error) {
```

Because `effects` is spelled with `V, M, W, C, G` rather than fresh `$` parameters, a batch of a different configuration does not match; that is the whole mechanism behind "the compiler rejects a mismatch".

8.3.7 Tagged unions and flat dispatch

`Message(Value)` is a tagged union of nine structs, and `node_step` dispatches on it with one `switch`:

src/consensus.odin

```
    switch msg in envelope.message {
    case Prepare_Message:       return on_prepare(node, envelope.from, msg, effects)
    case Promise_Message(V):    return on_promise(node, member, msg, effects)
    case Promise_Range_Message: return on_promise_range(node, member, msg, effects)
    case Accept_Message(V):     return on_accept(node, envelope.from, msg, effects)
    case Accepted_Message:      return on_accepted(node, member, msg, effects)
    case Commit_Message(V):     return on_commit(node, envelope.from, msg, effects)
    case Learn_Message:         return on_learn(node, envelope.from, msg, effects)
    case Nack_Message:          on_nack(node, msg)
    case Heartbeat_Message:     on_heartbeat(node, envelope.from, msg, effects)
    }
```

`member` is the sender's stable membership index, resolved once by `membership_index_of` before the switch; the handlers that count promises or acknowledgements take it instead of the id. A non-`#partial` switch over a union must name every variant, so adding a tenth message kind fails to compile until every dispatcher handles it. The `Write` union is consumed the same way by `ledger_apply`, `ledger_replay_fold`, and `effects_requires_power_loss_barrier`; `message_value` uses `#partial switch` deliberately, because most kinds carry no value.

8.3.8 `Maybe` for absent values

Optional state is a `Maybe(T)`, never a sentinel inside `T`. `leader_hint` is `Maybe(Node_Id)`, the remembered `noop` is `Maybe(Value)`, and a log's decided `stop_sign` is `Maybe(Stop_Sign(...))`. Unwrapping uses `.?`, either directly where absence is impossible or in the two-value form where it must be tested:

src/consensus.odin

```
    if node.role == .Leader {
        resend_to(node, peer, effects)
    } else if hint, ok := node.leader_hint.?; ok && hint == peer {
        request_learn(node, peer, effects)
    }
```

`node_current_leader` returns `node.leader_hint.?` unchanged, which is why its result is `(Node_Id, bool)`. Where absence is a normal state of a cell, the library uses an enum instead: `Cell_State` has an explicit `.Empty`, because a cell has three states, not two.

8.3.9 Bit sets, native and wrapped

Odin's native `bit_set` holds at most one machine word of members, and both windows and memberships may be larger, so `src/bit_set.odin` wraps an array of `bit_set[0..<64]` words. Insertion reports whether the member was new, which is what a vote counter needs:

src/bit_set.odin

```
// Inserts an index. Returns true when it was not present before.
bit_set_insert :: #force_inline proc(bs: ^Bit_Set($N), index: int) -> bool {
    w, b := index / WORD_BITS, index % WORD_BITS
    if b in bs.words[w] do return false
    bs.words[w] += {b}
    return true
}
```

A leader counts votes per slot in `acknowledgements[cell]`, a `Bit_Set(MAX_MEMBERS)`, and keeps the count in `acknowledged[cell]` so the quorum test is one comparison; a duplicate reply cannot inflate it:

src/consensus.odin

```
    if bit_set_insert(&node.acknowledgements[cell], member) do node.acknowledged[cell] +=
1
    if int(node.acknowledged[cell]) < membership_write_quorum(&node.membership) do
return .None
```

8.3.10 `core:container/small_array` for bounded lists

Every bounded list is a `small_array.Small_Array(N, T)`: the membership's members, a stop sign's members and metadata, an owner's resubmit queue, and the four lists in `Effects`. The storage is inline in the struct, `push_back` reports overflow as a `bool` instead of growing, and `slice` hands the host a view without a copy. The library's own append helpers write into the inline storage directly and treat an overrun as a bug rather than as backpressure (`effects_overrun` panics with a hint), because the capacity was computed to make overflow impossible:

src/effects.odin

```
effects_add_write :: #force_inline proc(e: ^Effects($V, $M, $W, $C, $G), w: Write(V)) {
    n := e.writes.len
    if n >= len(e.writes.data) do effects_overrun("writes")
    #no_bounds_check e.writes.data[n] = w
    e.writes.len = n + 1
    e.writes_pending = true
}
```

8.3.11 `or_return` and named results

A procedure that returns an `Error` last can propagate a callee's error with `or_return`, which keeps the happy path unindented. Named results make the propagation read as a sentence:

src/consensus.odin

```
) -> (slot: Slot, err: Error) {
    effects_reset(effects)
    proposal_gate(node) or_return
    if node.ownership do return propose_owned(node, value, effects)
    if node.next_slot == max(Slot) do return 0, .Global_Slot_Exhausted
    if node.next_slot - node.memory_floor > Slot(W) do return 0, .Window_Full
    slot = node.next_slot
    node.next_slot += 1
    send_accept(node, slot, node.ballot, value, effects) or_return
    return slot, .None
}
```

8.3.12 `for &record in` when the element's address matters

Iterating by reference makes it explicit at the loop header that the body takes the element's address or mutates it in place. The journal replay in the test harness needs the address, because the record it hands to the ledger must point at the journal's own copy of the value:

tests/harness.odin

```
    for &record in journal {
        write := record.write
        #partial switch &x in write {
        case paxos.Write_Vote(V):   x.value = &record.value
        case paxos.Write_Chosen(V): x.value = &record.value
        }
        paxos.ledger_replay_fold(ledger, write) or_return
    }
```

A loop written `for record in` copies each element, and `&record.value` would then point at a temporary that dies with the iteration; the `&` is the difference a reviewer looks for. Inside `src/`, scans over the ledger index the columns directly, `for cell in 0..<W`, because there is no element struct to take the address of.

8.3.13 `when INVARIANT_CHECKS` invariant checks

Structural invariants are asserted in every lifecycle procedure (`node_init`, `node_init_learner`, `node_begin_recovery`, and the shared `node_resume_at`), but only when `INVARIANT_CHECKS` is set, so the release benchmark measures the protocol rather than the checker:

src/node.odin

```
@(private)
node_assert_valid :: proc(node: ^Node($V, $M, $W, $C, $G)) {
    when INVARIANT_CHECKS {
        assert(node.id != 0, "node id cannot be zero")
        if node.voting_member {
            assert(membership_contains(&node.membership, node.id), "voter outside
membership")
        } else {
            assert(!membership_contains(&node.membership, node.id), "non-voter inside
membership")
            assert(!node.campaign_enabled, "non-voter cannot campaign")
        }
```

`INVARIANT_CHECKS` is `#config(PAXOS_INVARIANT_CHECKS, ODIN_DEBUG)` in `src/paxos.odin`: on in debug builds, off in release builds, and overridable either way with `-define:PAXOS_INVARIANT_CHECKS=true` or `=false`. `when` is a compile-time branch: the body is not compiled out by an optimiser's choice but omitted from the program in the first place. Nothing the host can observe depends on these checks, which is the test for whether a `when INVARIANT_CHECKS` guard is appropriate.

8.4 Control-Flow Rules

- **Return early.** A transition tests its preconditions first and returns the matching `Error`, so the mutation that follows is unconditional: `proposal_gate` opens `node_propose` with `.Not_Voter`, `.Not_Leader`, and `.Leader_Catching_Up` before a slot is assigned.
- **Dispatch flat.** One `switch` per union, one handler per variant, no handler table and no reflection.
- **Bound every loop.** A loop runs over a fixed array, a `small_array` slice, a bitmap, or a range no larger than a type parameter. `resend_to` iterates at most `WINDOW_SLOTS` times and stops after `CHUNK_SLOTS` sends, so one transition cannot emit more than the batch was sized for.
- **Never wrap.** Slot arithmetic goes through `slot_add`, which saturates at `max(Slot)`, and the counters in `tick` use `saturating_increment`.
- **No hidden allocation.** A transition receives its output buffer as a `^Effects` argument and never returns a slice it had to allocate.

8.5 Memory and Type Guidelines

- Types are `Ada_Case` (`Node_Options`, `Write_Vote`, `Durability_Gate`); procedures are `snake_case`.
- Procedures are prefixed with their receiver: `node_`, `effects_`, `membership_`, `ledger_`, `ballot_`, `bit_set_`, `replicated_log_`, `learner_`. The proc groups in `src/paxos.odin` add the short verbs on top; they do not replace the long names.
- `Node_Id :: u16` and `Slot :: u64` are plain aliases, not distinct types. The compiler will not stop a host from passing a slot where an id belongs, so the names must carry that weight at every call site. `Ballot :: distinct u64` is the exception, because a ballot is compared but never used as an index or a count.

- Zero is reserved as a sentinel wherever an identity or position is one-based: node id zero, slot zero, configuration id zero, and `BALLOT_ZERO`. `membership_init` rejects a zero id with `.Invalid_Node_Id`, and every transition that addresses a log entry rejects slot zero with `.Invalid_Slot`.
- Large values are written through destination pointers. `node_init` assigns `node^ = Node(V, M, W, C, G){...};` nothing returns a `Node` or a `Ledger` by value.

8.6 State and Invariant Safety

The ledger moves in one direction. `ledger_apply` refuses a lower promise, a second value under one ballot and slot, a second value for one chosen slot, and a record that would overrun a cell an earlier slot still holds; `install_chosen_trim` refuses an anchor that moves backward or disagrees with the adopted one under the same `trim_id`. The window is a ring, so a cell is claimed only when it is empty or its previous occupant is safe to forget: `claim_live` reuses a cell only below the memory floor and only if it held a decision. A vote above the floor is never overwritten by the library, because it may be part of a quorum that chose a value this node has not learned.

Every mutation of `node.ledger` is paired with the `Write` that will make it durable, in the same procedure, a few lines apart: `ledger_record_vote` with `Write_Vote`, `ledger_record_chosen` with `Write_Chosen`, an assignment to `promised` with `Write_Promise`. When reviewing, find the mutation and find the `effects_add_write`; if one is present without the other, the in-memory node and the journal will disagree after a restart.

8.7 Errors Versus Assertions

Three kinds of failure use three different mechanisms, and the choice is part of the design.

- **Return an `Error`** for any condition the host can cause or observe: a message from a non-member, a proposal on a follower, a full window, a journal record that regresses a promise. The host decides whether to retry, route elsewhere, or stop, and `explain_error` tells an operator which.
- **Assert** only what the library itself guarantees. The capacity check in `effects_add_write` and its three siblings in `src/effects.odin` (one branch, then `effects_overrun`) guards a capacity the library computed. If one of these fires, the bug is in `src/`, not in the host. The invariant checks in `node_assert_valid` are the same class, gated by `INVARIANT_CHECKS`.
- **Stop the process** only for the durability gate. `host_order_violation` is declared `-> !`, prints its banner, and calls `os.exit(1)`. The violation is the host's, but it cannot be returned as an error, because a host that reached this point has already shown it does not check the order; and it cannot be a debug assertion, because the danger is greatest in production.

`#assert` is the fourth tool, for anything decidable at compile time: capacity bounds, the power-of-two window, and the comparability of `Value`.

8.8 Comments and Documentation

Every public procedure carries a doc comment directly above it that states what the host must know: ordering, durability, bounds, or ownership. The comment on `node_is_leader_caught_up` is the model, because it says what the query does not promise as clearly as what it does:

src/node.odin

```
// Reports prefix catch-up only. This is not a lease and not a read barrier.
node_is_leader_caught_up :: proc(node: ^Node($V, $M, $W, $C, $G)) -> bool {
    return node.delivered_through >= node.leader_base - 1
}
```

Inside a procedure, a comment explains **why** a line exists, not what it does. `node_begin_recovery` explains why votes above the anchor survive; `on_heartbeat` explains why a follower promises to a ballot it never saw a prepare for; `record_commit` explains why a decision past the window edge takes the `pass_through` path. Unsupported behaviour is stated where a reader would look for it, as in the example above, rather than left for the reader to infer.

8.9 Tests and Measurements

`tools/check.py` is the repeatable verification entry point. It runs in a temporary directory so a stale binary can never mask a failure, and it performs, in order:

1. **Style**: every `.odin` file within the Zen constraints (108 columns, 1,408 lines, 70-line bodies), then `odin check -vet -strict-style` on `tests`, `sim`, `bench`, `cli`, and `examples/counter.odin`. The library is fully parametric, so its bodies are checked through the packages that instantiate it.
2. **Unit tests in two builds**: `odin test tests` with `-debug` and again with `-o:speed`, so a test cannot pass only because an invariant check was present.
3. **Contracts**: `tools/check_contracts.py`, the nine compile-fail fixtures and the four durability fixtures from the library chapter, the latter built with `-debug` and with `-o:speed`.
4. **Seeded simulations**: the `sim` binary for one, three, and five nodes across a range of seeds and steps (twenty seeds of ten thousand steps by default), once with a single leader and once with `--ownership`, with crashes injected inside the host commit sequence and oracles run after every transition. Another 120 runs use window 8/chunk 3 and majority or extreme flexible quorums.
5. **The example**: `examples/counter.odin` must run to completion.
6. **Benchmark schema**: the `bench` binary with `--iterations=1024 --json` must report eleven results with positive throughput and latency; the numbers themselves are not asserted.
7. **CLI failure propagation**: with a fake `odin` that exits non-zero on the path, `cli test` must fail and print a `Hint:` line, so a wrapper can never report success over a failed tool.

A protocol bug found by any of these gets a deterministic test of its schedule before it is fixed.

8.10 Review Checklist: A Human Proof Outline

Read a change to `src/` in this order. Each question is a proof obligation, and a change that cannot answer one is not ready.

1. **Which invariant does this touch?** Name it: B1 ballot uniqueness, B2 quorum intersection, B3 max-vote preservation, D1 indelible ink, L1 contiguous delivery, or S1 stop-sign sealing, as listed at the top of `src/paxos.odin`.
2. **Is every ledger mutation paired with its `Write`?** Find the `ledger_record_vote`, `ledger_record_chosen`, or assignment to `promised` or `promised_at`, and the matching `effects_add_write`.
3. **Does the message leave only after the write?** The write is appended to the batch before the message, and the host order does the rest; a new message that depends on a new write must follow the same pattern.
4. **Does every pointer in the batch point at something that outlives the batch?** Into the ledger, or into `pass_through`; never at a local.
5. **Is the handler idempotent?** Deliver the message twice, and once late. A duplicate `Accepted_Message` must not count twice; a stale `Prepare_Message` must draw a `Nack_Message`, not a promise.
6. **Is every loop bounded?** By a fixed array, a `small_array` length, a bitmap, or a type parameter; and is the batch capacity still the exact maximum?
7. **Are the bitmaps in step with the columns?** A cell that becomes `.Voted` or `.Chosen` is inserted into `used` (and `chosen`); a cell that is reopened is removed from both.
8. **Does zero still mean what it did?** A new field's zero must be a valid state, and a new option's zero must be the default.
9. **Is the failure classified correctly?** Host-caused conditions return an `Error` with a banner; library guarantees are asserted; only the gate stops the process.
10. **What does restore do with it?** If the change adds durable state, both `ledger_apply` and `ledger_replay_fold` must fold it, and `node_resume_at` must recompute any frontier derived from it.
11. **Which test reproduces the bug or exercises the feature?** A unit test with a fixed schedule, and where delivery order matters, a simulator oracle in both leadership modes.
12. **Can a colleague read the diff as a proof?** Precondition, protected state, durable change, emitted evidence, duplicate behaviour, failure behaviour, in that order, without opening this book.

Review & Discussion
Code Review Exercise: Select `on_accept` or `ledger_apply` and outline its proof structure:

- Preconditions and parameter validation.
- Invariants and protected ledger state.
- Durable writes emitted to the caller's batch.
- Idempotency and behaviour on duplicated messages.
- Error diagnostics, hints, and remediation paths.

PART V

Three worked systems

We run the repository's counter line by line, design a key-value host around the library's contract, and rehearse a partition in a five-voter control plane. The guidance thins as the systems grow.

9 Three Worked Systems

Learning Objectives
After completing this chapter, you will be able to:
- Trace the event-driven host loop and output of the runnable replicated counter.
- Implement durable client request deduplication and read-consistency disciplines in a key-value store.
- Evaluate architectural conditions that justify enabling rotating slot ownership.
- Formulate precise pass/fail criteria and operational procedures for regional partition drills.

9.1 Small Example: The Replicated Counter

`examples/counter.odin` is the complete integration contract in miniature. It runs three nodes in one process, with a queue standing in for the network and nothing standing in for the journal. Run it from the repository root with `odin run examples/counter.odin -file`.

9.1.1 The command and the types

examples/counter.odin

```
package main

import "core:container/queue"
import "core:fmt"
import paxos "../src"

Command :: struct {
    client_id:  u32,
    request_id: u32,
    amount:     i64,
}

MEMBERS :: 3
WINDOW  :: 64
CHUNK   :: 16

Node    :: paxos.Node(Command, MEMBERS, WINDOW, CHUNK)
Effects :: paxos.Effects(Command, MEMBERS, WINDOW, CHUNK)
```

`Command` is three integers, so it is comparable and self-contained. The two aliases pin one configuration, three members with a 64-slot window and 16-slot recovery chunks, and share it between node and batch, which is what lets `paxos.step(&node, envelope, &effects)` type-check. `WINDOW` must be a power of two, because a slot's cell is `(slot - 1) & (WINDOW - 1)`; `node_init` rejects any other value at compile time.

9.1.2 The packet: a value copied at enqueue

examples/counter.odin

```
// An envelope in flight. A message points at a value inside the sender's ledger, so a
// transport copies the value when it queues the envelope, exactly as a codec would.
Packet :: struct {
    envelope: paxos.Envelope(Command),
    value:    Command,
}
```

```
packet_of :: proc(envelope: paxos.Envelope(Command)) -> (packet: Packet) {
    packet.envelope = envelope
    if value, carries := paxos.message_value(envelope.message); carries do packet.value
= value^
    return
}

packet_envelope :: proc(packet: ^Packet) -> paxos.Envelope(Command) {
    envelope := packet.envelope
    #partial switch &m in envelope.message {
    case paxos.Promise_Message(Command): m.value = &packet.value
    case paxos.Accept_Message(Command):  m.value = &packet.value
    case paxos.Commit_Message(Command):  m.value = &packet.value
    }
    return envelope
}
```

This is the one idiom the data-oriented core asks of every host. A `Promise_Message`, `Accept_Message`, or `Commit_Message` carries its value as a `^Command` that points into the sending node's ledger, and that pointer is valid only until the sender's next transition. A real transport serialises the value into a frame at that moment; this in-process transport does the equivalent by copying it into the `Packet`. `paxos.message_value` answers whether the message kind carries a value at all, so the variants without values are copied unchanged. `packet_envelope` reverses the move before delivery: it repoints the message at the packet's own copy, which the caller keeps alive for the duration of `step`. The test harness (`tests/harness.odin`) and the simulator carry the same two procedures under the same names.

9.1.3 The cluster

examples/counter.odin

```
Cluster :: struct {
    nodes:   [MEMBERS]Node,
    network: queue.Queue(Packet),
    counter: i64,
}
```

The network is a `core:container/queue` of packets; this host delivers every message once and in order, and the fault simulator in `sim/` is where drops, duplicates, and crashes live. `counter` is the application state, one for the whole cluster, for a reason the output section explains.

9.1.4 The host commit sequence

examples/counter.odin

```
// Consumes one node's effects in the order the durability contract requires.
host_commit :: proc(cluster: ^Cluster, node_index: int, effects: ^Effects) {
    // 1. Append effects.writes to a journal and sync it. This example keeps no journal.
    // 2. Tell the batch its writes are durable; only then may messages leave.
    paxos.confirm_writes_durable(effects)
    // 3. Transmit.
    for envelope in paxos.messages_slice(effects) {
        queue.push_back(&cluster.network, packet_of(envelope))
    }
    // 4. Apply newly decided entries, in slot order. One node narrates.
    for entry in paxos.committed_slice(effects) {
        if node_index == 0 {
            cluster.counter += entry.value.amount
            fmt.printfln("slot %d: %+d -> counter = %d", entry.slot, entry.value.amount,
```

```
cluster.counter)
        }
    }
}
```

This is the contract from the bounded-core chapter with step 1 reduced to a comment. The order is still real: if `messages_slice` came before `confirm_writes_durable` the program would stop with the durability banner. A real host replaces the comment with an append and a sync, and `queue.push_back` with a send. Note that step 3 copies each value out of the ledger through `packet_of` before the node moves on, and that `entry.value` in step 4 is a `^Command` into the same ledger, read here before the next transition.

9.1.5 Settling the network

examples/counter.odin

```
// Delivers every queued envelope until the network is silent.
settle :: proc(cluster: ^Cluster) {
    effects: Effects
    for packet in queue.pop_front_safe(&cluster.network) {
        packet := packet
        envelope := packet_envelope(&packet)
        to := int(envelope.to - 1)
        err := paxos.step(&cluster.nodes[to], envelope, &effects)
        assert(err == .None, paxos.explain_error(err))
        host_commit(cluster, to, &effects)
    }
}
```

`settle` pops packets until the queue is empty, rebuilds the envelope over the packet's own copy of the value, steps the addressee, and commits that node's batch, which may push more packets. The shadowing `packet := packet` gives the loop variable an address that outlives the `step` call. One `Effects` value serves every step: each transition resets it, and each reset succeeds because `host_commit` confirmed the previous batch.

9.1.6 `main`: election, then three proposals

examples/counter.odin

```
    membership: paxos.Membership(MEMBERS)
    ids := [MEMBERS]paxos.Node_Id{1, 2, 3}
    // Keep side effects out of assert: a release build may compile assertions away.
    membership_err := paxos.init(&membership, ids[:])
    assert(membership_err == .None)
    for &node, i in cluster.nodes {
        node_err := paxos.init(&node, ids[i], membership, paxos.Node_Options{priority
= u8(i)})
        assert(node_err == .None)
    }

    // Node 1 runs phase one once; every later command commits in one round trip.
    effects: Effects
    noop := Command{}
    campaign_err := paxos.campaign(&cluster.nodes[0], noop, &effects)
    assert(campaign_err == .None)
    host_commit(&cluster, 0, &effects)
    settle(&cluster)
    assert(paxos.role(&cluster.nodes[0]) == .Leader)
    fmt.println("node 1 is the leader")
```

`paxos.init` is called with two receivers: the membership (a slice of ids, majority quorums by default) and each node. Every call that has an effect is bound to a variable first and asserted afterwards, because `assert` may vanish from a release build and take its argument with it. The priorities 0, 1, and 2 (a `u8`) only break ties between campaigns in the same round; nothing here calls `tick`, so no timeout fires and node 1 is the only candidate. Its `Prepare_Message` goes to all three nodes, itself included; `settle` delivers the promises back, and once a read quorum of two has described the empty chunk node 1 is leader.

examples/counter.odin

```
    commands := [?]Command{
        {client_id = 1, request_id = 101, amount = 10},
        {client_id = 1, request_id = 102, amount = 25},
        {client_id = 2, request_id = 201, amount = -5},
    }
    for command in commands {
        slot, err := paxos.propose(&cluster.nodes[0], command, &effects)
        assert(err == .None, paxos.explain_error(err))
        fmt.printfln("proposed request %d in slot %d", command.request_id, slot)
        host_commit(&cluster, 0, &effects)
        settle(&cluster)
    }

    // Every node holds the same decided log.
    for &node in cluster.nodes {
        assert(paxos.decided_through(&node) == len(commands))
        for slot in 1..=paxos.Slot(len(commands)) {
            value, ok := paxos.committed_at(&node, slot)
            assert(ok && value == commands[slot-1])
        }
    }
    fmt.printfln("counter = %d on all %d nodes", cluster.counter, MEMBERS)
    assert(cluster.counter == 30)
```

Prediction Exercise

Node 3 never proposes and never narrates. Before reading the output, write down what `paxos.decided_through(&cluster.nodes[2])` returns after the loop, and which message kind delivered each of its three decisions.

9.1.7 The output

```
node 1 is the leader
proposed request 101 in slot 1
slot 1: +10 -> counter = 10
proposed request 102 in slot 2
slot 2: +25 -> counter = 35
proposed request 201 in slot 3
slot 3: -5 -> counter = 30
counter = 30 on all 3 nodes
```

9.1.8 Why the counter ends at 30 on all three nodes

The arithmetic is $10 + 25 - 5 = 30$. The claim about three nodes needs the trace of one proposal. `propose` on node 1 assigns the next slot, records node 1's own vote in its ledger as a `Write_Vote`, counts itself in the slot's acknowledgement set, and broadcasts `Accept_Message` to nodes 2 and 3. Each peer records its vote, again as a `Write_Vote`, and replies with `Accepted_Message`. When the first reply reaches node 1 the acknowledgement set holds two distinct members, which meets the write quorum of two, so node 1 records a `Write_Chosen`, releases the entry through `committed_slice`, and broadcasts `Commit_Message` to both peers. Each peer records the decision on

top of its vote and releases the same entry. The second `Accepted_Message` arrives at a cell whose state is already `.Chosen` and changes nothing.

Only node index 0 adds to `cluster.counter`, so the number printed is node 1's view. The final loop extends it to the other two: every node reports `decided_through` of 3, and `committed_at` on every node returns the very `Command` proposed for that slot. The counter is a fold over the log, and three equal logs fold to the same 30; the example asserts that rather than assuming it.

The `client_id` and `request_id` fields are carried but never used. Request 101 and request 102 come from the same client, and nothing in this program would stop a retried 101 from being applied twice.

> **Exercise 16.1**
> Extend the counter so a client that retries a request after a timeout can never be applied twice. Say what the state machine stores and what the host returns to the client.

9.2 Middle Example: A Key-Value Host

This section is a design, not repository code. The library orders commands; the host owns the store, the client protocol, the snapshot, and the journal. The sketches below use those host types by name without defining them.

9.2.1 A bounded request discipline

A write may commit even though its reply is lost, so a client that times out does not know whether its command was applied. The host resolves that ambiguity inside the state machine, where it is replicated. Give every client a stable id and increasing request ids, allow one outstanding request per client, and store next to the data a table from client id to the last applied request id and its result:

```
Client_Record :: struct {
    request_id: u64,
    result:     Result,
}

State :: struct {
    values:       Bounded_Map(Key, Value_Hash),
    clients:      Bounded_Map(Client_Id, Client_Record),
    applied_slot: paxos.Slot,
}
```

The apply procedure advances through a duplicate's slot even when it suppresses the duplicate's effect, because the slot was decided whether or not the command was new:

```
apply :: proc(state: ^State, slot: paxos.Slot, command: Command) -> Result {
    assert(slot == state.applied_slot + 1)
    if record, known := state.clients[command.client_id]; known {
        if command.request_id < record.request_id {
            state.applied_slot = slot
            return stale_request_result()
        }
        if command.request_id == record.request_id {
            state.applied_slot = slot
            return record.result
        }
    }
    result := execute(state, command)
    state.clients[command.client_id] = Client_Record{command.request_id, result}
    state.applied_slot = slot
    return result
}
```

`stale_request_result` is a host-defined result for an older request whose answer is no longer retained. Returning the most recent answer for that older request would be incorrect. Keeping one answer per client is sufficient only with the stated one-outstanding-request discipline; hosts needing older answers retain more history.

`values`, `clients`, and `applied_slot` are persisted together and included in every snapshot; a dedup table that outlives the store would answer a retry with a result the store never held. The host calls `apply` from step 4 of its commit sequence with `entry.value^`, copying nothing it does not keep.

9.2.2 Timeouts do not mean failure

A client that times out retries the same request id, possibly at a different node. Three outcomes are possible, and the host handles each without guessing:

- The original committed. The retry is proposed, decided in a later slot, and `apply` returns the stored result without touching `values`.
- The original never committed. The retry is proposed and applied once.
- The node is not the leader. `paxos.propose` returns `.Not_Leader` before producing any effects; the host answers the client with the hint from `paxos.current_leader`, if there is one, and the client tries there.

`.Window_Full` and `.Leader_Catching_Up` are handled the same way as `.Not_Leader`: they describe a state to wait out, not a failed request.

9.2.3 Reads are an application protocol

`paxos.committed_at` and `paxos.read_decided` inspect the local log. They do not prove that this node is still leader, or that no later slot has been decided elsewhere, at the moment of the read. The library says so on `node_is_leader_caught_up`: it "is not a lease and not a read barrier".

A linearizable read on the leader therefore needs a barrier. The plain design is to propose a read-only command, capture the result from `values` when that command is applied, and return it only after local application. With replies sent only after application, the read follows every write that completed before the read began. This costs one consensus round per read, or per batch of reads that share a barrier. The cheaper alternative, a leader lease bounded by ticks and heartbeat acknowledgements, is not implemented in this repository; it is the subject of the proposal in `docs/pod/records/0004-fast-path-leases.typ`, whose status is "Open for Discussion". A follower may serve a read without a barrier only when the service contract calls it stale and the reply carries `applied_slot`, so the client knows which prefix was read. That index alone does not bound staleness in time.

9.2.4 Snapshots and `install_chosen_trim`

The window is finite, and `advance_memory_floor` only lets cells be reused; the journal and the peers still expect the history to exist somewhere. Periodically the host writes a state image of `State` through some applied slot S, syncs it, and then tells the node that everything at or below S is chosen and covered by that image:

```
anchor := paxos.Trim_Anchor{
    trim_id          = next_trim_id,
    chosen_trim_slot = image_slot,
}
err := paxos.install_chosen_trim(&node, anchor, &effects)
```

`Trim_Anchor` has exactly those two fields. The core compares anchors by identity and order only; it never hashes the image. The host therefore binds the image's checksum to `trim_id` on its own side, in a table from `trim_id` to the image's checksum and path, written and synced before the anchor is installed. When a peer's `Promise_Range_Message` later reports an anchor with a `trim_id` this host does not hold, the host knows it must fetch that image, and when it does hold the id it can verify the fetched bytes against the checksum it recorded.

`install_chosen_trim` resets the batch first, then returns `.Invalid_Slot` if the anchor lies above `decided_through`, `.Trim_Regression` if `trim_id` moves backward, if the same `trim_id` names a different slot, or if `chosen_trim_slot` moves backward, and `.None` with no write when the identical anchor is installed twice. Otherwise it emits a `Write_Trim`, adopts the anchor, and raises the memory floor to the anchor (capped at `decided_through`). From then on the node answers `Prepare_Message` with the anchor in its `Promise_Range_Message`, reports no cell at or below it, and a new leader fences its recovery above it. A node that has fallen below a peer's anchor cannot catch up from messages alone: `record_commit` ignores slots at or below its own anchor, and `on_accept` ignores accepts for them. The host installs the image and calls `paxos.begin_recovery(&node, anchor)`, which applies the anchor to the ledger through `ledger_apply`, keeps

the node's votes and decisions above it, returns the node to `.Follower`, and persists nothing on its own; the host must have the image and the anchor durable before running further transitions.

9.2.5 Serving `Serve_Range_Request` from the journal

When a peer's `Learn_Message` asks for slots at or below this node's memory floor, `on_learn` emits a `Serve_Range_Request{peer, first, count}` and the host answers from what it kept. The simulator keeps an applied array per node and does this:

sim/simulation.odin

```
    // Serve evicted history from the host's durable application image.
    for request in paxos.requests_slice(effects) {
        switch r in request {
        case paxos.Serve_Range_Request:
            for offset in 0..<r.count {
                slot := r.first + paxos.Slot(offset)
                if slot >= MAX_SIM_SLOTS do continue
                if value, ok := &sim.applied[node_idx][slot].?; ok {
                    commit := paxos.Commit_Message(u64){slot = slot, value = value}
                    enqueue(sim, paxos.Envelope(u64){from = node_id, to = r.peer, message
= commit})
                }
            }
        }
    }
```

A key-value host does the same from its journal of `Write_Chosen` records: each served slot goes back as a `Commit_Message` whose `value` points at the host's own copy for as long as the transport needs it, and the peer's `step` handles it as it would a leader's commit. Slots below the host's own image need the image.

9.2.6 When the host would choose rotating ownership

Everything above assumes one leader, which is the right shape when writes come from one region: a single phase one, then one round trip per command. A store whose writers sit in several regions pays a cross-region hop for every write that did not originate where the leader lives. For that host the library offers a second shape, selected per node at initialisation:

```
err := paxos.init(&node, id, membership, paxos.Node_Options{rotating_ownership = true})
```

Under rotating ownership there is no leader. Member i in membership order owns every slot s with $(s-1) \bmod N = i$ (`owner_of`), and proposes in its own slots at round zero (`ownership_ballot`) with no phase one at all, because no lower ballot exists in that decree. `campaign` answers `.Campaign_Disabled`, and `propose` succeeds on every member, so each region writes locally and pays one round trip. The price is contiguity: the log advances only as fast as its slowest owner. An idle owner fills its own slots below the highest slot it has seen with the no-op (at most `SKIP_BURST` per tick), and a stalled owner is repaired after `election_timeout_ticks` by a bounded revocation, a phase one over the stalled chunk that fences the owner out of those slots only and re-proposes any vote it finds. An owner whose suggestion lost to a revocation queues a best-effort resubmission; the host still handles retries and deduplication. The read command must be evaluated at its position in the applied sequence. Reply to writes only after contiguous application: a value merely chosen in a higher slot is not yet a completed application operation. Under that contract, the read follows every write completed before it began. The host picks this option when write latency across regions is the cost that matters, and keeps the single leader when a quiet region would otherwise be skipped for every slot it owns.

9.3 Large Example: A Regional Control Plane

This is an architecture and a drill, not runnable code. Assume five voters in three zones: nodes 1 and 2 in zone A, nodes 3 and 4 in zone B, node 5 in zone C. The membership is `paxos.Membership(5)` with the default majority quorums, so any three voters can elect and commit, and the loss of any one zone leaves at least three.

Placement is expressed through `Node_Options`. If zone A is the preferred home for the leader, its nodes are initialised with `paxos.Node_Options{priority = 2}`, zone B with `paxos.Node_Options{priority = 1}`, and node 5 with `paxos.Node_Options{campaign_disabled = true}` so that a lone zone never leads but still promises and votes. Priority breaks ties between campaigns in the same round; a campaign in a higher round wins regardless, as the `.Campaign_Disabled` banner reminds the reader. A longer `election_timeout_ticks` in the less preferred zones reduces the number of simultaneous campaigns after a leader fails, at the cost of a slower failover when the preferred zone is the one that failed.

9.3.1 The partition drill

Node 1 leads. The operator isolates zone A from zones B and C; the link between nodes 1 and 2 stays up.

Step	Actor	Event and reason
1	Operator	Cuts every link between zone A and the rest. Node 1 is still `.Leader` in memory; its heartbeats now reach only node 2.
2	Node 1	Accepts a client command and proposes it. `send_accept` records its own `Write_Vote`; the `Accept_Message` reaches node 2 only, so the slot's acknowledgement set holds two members against a write quorum of three. Nothing commits. The client sees a timeout, which is not a failure.
3	Nodes 3, 4, 5	Count `election_timeout_ticks` ticks without leader contact. Node 3 or 4 campaigns (node 5 is `campaign_disabled`); `start_campaign` takes a round one above the highest it has observed, so its ballot exceeds node 1's, and its `Prepare_Message` names `first = decided_through + 1` with `scope = .Global`.
4	Nodes 3, 4, 5	Each records a `Write_Promise`, syncs, and only then sends its `Promise_Messages` and the closing `Promise_Range_Message`. Three answers meet the read quorum. No promiser voted in the slot node 1 was filling, so `resolve_chunk` drives nothing and `become_leader` sets `next_slot` to that slot: the new leader's first proposal takes it. Node 1's value was never chosen.
5	New leader	Commits with votes from 3, 4, and 5. Node 1 still believes it leads: it accepts proposals that never commit, and once `WINDOW_SLOTS` of them are open `propose` returns `.Window_Full`. Zone A serves no writes that anyone will ever read back.
6	Operator	Heals the links.
7	Node 1	Receives a `Heartbeat_Message` with the higher ballot, records a `Write_Promise` in `on_heartbeat`, and drops to `.Follower`; or its own stale heartbeat draws a `Nack_Message`, and `on_nack` has the same result and records `ballot_node(promised)` as the leader hint. Its clients now get `.Not_Leader` and the new leader's id from `current_leader`.
8	Nodes 1, 2	See a `decided_through` above their own in the heartbeat and send `Learn_Message`; the leader answers with `Commit_Message` for each slot. Node 2's stale vote is superseded by the decision `record_commit` writes on top of it as a `Write_Chosen`. The host may also call `paxos.reconnected` on each side to start the repair immediately.
9	Leader host	If the outage outlasted the leader's memory floor, the `Learn_Message` produces a `Serve_Range_Request` and the host serves from its journal. If it outlasted the trim anchor, zone A's hosts fetch the image named by the anchor's `trim_id`, verify it against the checksum they recorded, install it, and call `begin_recovery` before those nodes vote again.

9.3.2 Exit criteria

The drill passes on observations, not on the absence of alarms.

Observation	Pass	Fail
Writes in the isolated zone	Every proposal in zone A times out; none is later reported as applied.	A command voted for only by nodes 1 and 2 appears in the majority side's decided log.
Election on the majority side	A zone B node becomes `.Leader` with a round above node 1's.	No leader within the timeout, or node 5 leads.
Log agreement after healing	`decided_through` is equal on all five nodes and `committed_at` agrees slot by slot.	Two nodes report different values for one slot, or any node returns `.Conflicting_Commit`.
Step-down	Node 1 reports `.Follower` and its hint names the new leader.	Node 1 still reports `.Leader` after receiving the new ballot.
Catch-up path	Every `Serve_Range_Request` was answered from the journal; every below-anchor node was rebuilt from the image its `trim_id` names.	A node below the anchor votes before its image is installed.
Durability banner	Never printed.	Printed once, by any node.

Exercise 17.1
Write the exit criteria for the regional partition drill: what must be observed before the drill counts as passed, and which observation would fail it.

Review & Discussion
Classify system responsibilities for a multi-region deployment across three domains:

- Core Guarantees: Monotonic consensus order, epoch isolation, and durability verification.
- Host Obligations: Disk synchronization, network transport, payload copying, and snapshot transfer.
- Operational Policies: Leader locality, client routing, and request deduplication windows.

PART VI

Evidence

A proof obligation, a unit test, a seeded fault run, and a benchmark answer four different questions. This part says which question each one answers, what the repository actually runs today, and what none of it can tell you.

10 Validation, Testing, and Operations

Learning Objectives
After completing this chapter, you will be able to:
- Classify the specific guarantees provided by formal proofs, unit tests, deterministic chaos simulation, and benchmarks.
- Execute the full verification harness and interpret diagnostic outputs.
- Replay and debug failing distributed schedules using deterministic simulation seeds.
- Evaluate comparative benchmark results without conflating CPU throughput with end-to-end service latency.
- Design operational chaos drills with rigorous pass/fail criteria.

Checkpoint: Vocabulary Check
Ensure precise command of these three distinct events:
- **Chosen:** A write quorum has durably accepted a value under a ballot.
- **Committed:** A node has received evidence that a value is chosen (or certified it locally).
- **Applied:** The host state machine has executed the command in sequence.

The simulator oracles below verify invariants across each of these boundaries.

10.1 Four kinds of confidence

Evidence	Question it answers	What it cannot answer
1. Safety argument (Parts I-III, the lemmas in the safety-argument chapter)	Why every legal transition preserves agreement, and why each departure from the textbook keeps the theorem.	Whether the Odin code and the host actually follow the argument.
2. Deterministic unit tests (`tests/`)	Whether specific schedules, including reordered and duplicated messages, produce the required state.	Whether unvisited interleavings are safe.
3. Seeded fault simulation (`sim/`, `tests/test_reconfiguration_sim.odin`)	Whether agreement,	Whether a real disk lies

Evidence	Question it answers	What it cannot answer
	validity, monotonicity, contiguity, and convergence survive thousands of random crashes, drops, duplicates, and partitions, with one leader and with every node proposing in its own slots.	about durability, or whether a real network authenticates peers.
4. Benchmarks (`bench/`)	How much CPU one committed value costs in this process, and what one storage barrier costs on this disk.	Service latency on a real network, or how the libraries compare on hardware other than the recording host.

Each row is necessary and none is sufficient. A passing simulation is finite executable evidence; it is not a proof, and this book never calls it one. No model-checked specification ships with this repository.

10.2 What the repository tests today

`odin test tests` runs 79 deterministic tests. They are grouped below by file, with representative procedure names where a particular rule needs a direct reference.

File	Tests	What is pinned down
`test_protocol.odin`	7	Membership validation, ballot ordering, single-node and three-node consensus, the proc-group surface, restore and `continue_at`, and Lamport's greatest-vote rule (`test_lamport_b3_max_vote_rule`).
`test_election_matrix.odin`	1 (972 cases)	Every assignment of no vote / ballot 1 / ballot 2 to three voters, every first-response order, and all six intersecting quorum pairs: if an earlier write quorum chose a value, every later decision preserves it.

File	Tests	What is pinned down
`test_review.odin`	21	Regressions found in review: campaigns discard prior-term proposals, fences survive chunk boundaries, retries make progress across chunks, snapshots keep votes above the anchor, trim identity conflicts fail closed, duplicate acknowledgements never make a quorum, 128 and 1,024 voters reach a quorum through the sorted membership array, a leader fetches decisions from a follower that is ahead, and more.
`test_ownership.odin`	5	Rotating ownership: three owners decide concurrently without a campaign, idle owners skip, a crashed owner's slots are revoked, a revocation keeps a vote it finds (Lamport's B3), and a suggestion revoked to the no-op is resubmitted in a later own slot.
`test_window_review.odin`	8	The second adversarial review: a follower refuses slots beyond its window, the pass-through releases one decision per transition with its own value, a stale acknowledgement is not an error, an owner keeps proposing after the floor passes its next slot, a far accept cannot wedge an owner, a suggestion the owner itself overwrites is resubmitted, a revocation range stays inside the window, and a leader whose inherited gap stalls re-runs phase one.
`test_batch_review.odin`	6	The third adversarial review: an ownership tick fits the effect capacities under write quorum one, an ownership batch is admitted on the owner's own frontier after the floor advances, a rejected batch leaves nothing behind, an older vote reported after a decision is not a conflict, ownership order is ascending id whatever order the host gave, and a resubmission the bounded queue cannot hold is counted.
`test_recovery_chunk.odin`	7	Chunk sizes 1, 3, and 8; ring crossings; reordered and duplicate reports; stale reports; window backpressure; large payloads; frozen phase-two selection; sparse retransmission without duplicate sends.
`test_slot_exhaustion.odin`	3	Boundary behaviour near the largest representable slot.
`test_reconfiguration.odin`	2	A three-node handover with a delayed old-configuration message rejected by the checked `Log_Envelope` step; stop-sign initialisation with aliased slices.
`test_reconfiguration_sim.odin`	4 (16 seeds each)	Seeded, shuffled delivery on a replicated log: a seal survives a dropped, a duplicated, and a re-

File	Tests	What is pinned down
		ordered accept; a 1,2,3 to 2,3,4 handover; a one-for-one voter replacement; under rotating ownership, decisions other owners reach above the stop sign are abandoned and re-decided by the next configuration. Each run checks seal agreement, nothing released past the seal, replay keeps the seal, and the next configuration decides new commands on the same slot line.
`test_replicated_log.odin`	3	Commands, a stop sign that seals the epoch, and the handover initialisers.
`test_learner.odin`	3	Contiguous release, window wrap with `Trimmed` and `Window_Full`, configuration mismatch.
`test_durability.odin`	4	`requires_power_loss_barrier`, the pre-durable accept iterator, the host-managed gate, and a zero-initialised batch being ready without an init call.
`test_errors.odin`	2	Every `Error` value has a title, an explanation, and a `Hint:`; adding a value without one fails the build's tests.
`test_bit_set.odin`	3	The bounded slot set and the native `bit_set` operators.

Two checks run outside `odin test` because they need separate processes. The script `tools/check_contracts.py` compiles nine programs that must be **rejected**: a zero member capacity and one of 65,536, a zero window and a window that is not a power of two, a zero chunk and a chunk larger than the window, a zero learner window, a non-comparable `Value`, and an `Effects` whose parameters differ from its `Node`. It then builds four programs in both debug and optimized modes: two must stop with a `DURABILITY ORDER VIOLATION` diagnostic (reading messages before confirming writes; resetting a batch with unconfirmed writes), and two must exit cleanly (the correct order; a zero-initialised batch used without any init call).

10.3 The deterministic fault harness

`sim/simulation.odin` drives one to five voters from a single seed. Every choice comes from a SplitMix64 generator, so a failure is replayed exactly by the command the failure prints. With `--ownership` every node is started with rotating ownership and proposes in its own slots; the same oracles apply, and the liveness probe may be answered by any node.

shell

```
./bin/paxos-sim --seed=1337 --steps=10000 --nodes=5 --verbose
./bin/paxos-sim --seed=1337 --steps=10000 --nodes=5 --ownership
```

Each step rolls one action:

Share	Action	Faults applied
45%	Deliver one queued envelope	Dropped at 6% (default), duplicated at 4%, blocked by a cut link, or lost because the target is down.
20%	Tick a live node	Elections, heartbeats, retransmission.
15%	Propose at a random live node	One in four proposals is a two-value batch. `Not_Leader`, `Window_Full`, and `Leader_Catching_Up` are expected backpressure.

Share	Action	Faults applied
6%	Cut or heal one link	Asymmetric partitions accumulate.
4%	Crash a node	Only while more than a read quorum stays alive.
6%	Restart a crashed node	Journal replay with `ledger_replay_fold`, then `restore` at the host's consumed floor.
4%	Report a reconnected peer	`reconnected` triggers retransmission or a catch-up request.

The host side of every transition is itself a fault site. With probability `crash_permille` the process dies at one of three points of its commit sequence: before any write, after a random durable prefix of the writes with no message sent, or after every write with only a prefix of the messages sent. Accept requests at a campaign ballot may leave before the barrier, exactly as `pre_durable_messages` permits; an owner's round-zero suggestion may not, and it was this harness, at the vote level, that showed why: a restarted owner reused its ballot for a different value until the exception was narrowed. The memory floor is advanced only half of the time so that full-window and cell-reuse paths run.
The oracles run after every observed transition:

Oracle	What it rejects
Agreement	A durable decision for a slot that differs from the first durable decision for that slot. Decisions are observed at the **vote** level: a write quorum of identical durable votes counts, whether or not any leader announced it.
Validity	A decided value that is neither the no-op nor a value some node proposed.
Promise monotonicity	A `Write_Promise` below an earlier promise, or a `Write_Vote` below the current promise, on the same node.
Contiguity	A node releasing slot s before slot $s - 1$.
Liveness probe	After all faults stop, the healed cluster must decide one fresh proposal; a run that never decided anything cannot pass vacuously.
Convergence	After quiescence every node must have applied every slot of the golden log.

`make check` runs 240 simulations of ten thousand steps: 120 with the default window (one, three, and five voters; twenty seeds; both leadership modes), and 120 with an eight-slot window and three-slot chunks (three voters; majority, read-all/write-one, and read-one/write-all quorums; twenty seeds; both modes). It also checks style, tests in both build modes, contract fixtures, the example, the benchmark's JSON contract, and CLI failure propagation. The recovery review used `python3 tools/check.py --seeds=100`: 600 default-window runs plus the 120 focused runs, for 7.2 million steps. The focused matrix caps each configuration at twenty seeds. These are two different runs; the larger count is recorded evidence, not the default of `make check`.

> **Prediction Exercise**
> The simulator crashes a node after a random **prefix** of its writes has been journaled. Why can recovery safely replay that prefix if no reply relying on a lost promise or vote was sent? Contrast this with sending Accepted before its vote was durable.

10.4 Matched CPU measurements

First decide what a row means. The matched drivers in `bench/matched/` compare the cost of completing an ordered stream in memory. Every library processes the same 4,096 values per epoch, with the same voter count, payload size, and limit on outstanding work. Every learner's ordered payloads are checked. A separate warm-up precedes timing; repeated runs rotate the execution order. There is no disk, serialization, or network delay in this measurement.
The matrix covers three and five voters, 8-, 64-, and 1,024-byte values, and depths 1, 8, and 64. The four implementations retain their native algorithms: OmniPaxos can coalesce entries, and LibPaxos3 performs phase-

one preexecution. Equal workload does not mean identical work inside each implementation. The Odin baseline preserves the source from before the recovery-storage changes.

Recorded 20260917T073751Z on AMD Ryzen 7 5800H with Radeon Graphics. Nine samples per row; median ns per completed value; lower is better.

N	Bytes	Depth	Odin before	Odin	Zig	OmniPaxos	LibPaxos3
3	8	1	107.5	109.2	114.4	1062.1	2543.6
3	8	8	111.5	111	119	224.3	2548.5
3	8	64	108.1	111	118.3	88.1	2555
3	64	1	118.4	115.8	150.4	1228.7	2507.6
3	64	8	122.6	123.2	149.4	321.5	2586.7
3	64	64	119	121	152.4	96.3	2603
3	1024	1	357.1	300	1725.8	3251.8	3388
3	1024	8	430.3	338.1	1708.8	2246.7	3536.5
3	1024	64	469.9	349.3	1938	2450.5	3815.1
5	8	1	202.7	206.7	166	3077.3	3335.9
5	8	8	206.3	211.3	173.8	511.9	3348.7
5	8	64	204.5	209.5	177.5	153.2	3431.9
5	64	1	218.9	224.3	245.6	3269.3	3325.1
5	64	8	221.2	221.4	252.5	672.1	3404.5
5	64	64	218.2	220.8	259.5	250.4	3557.6
5	1024	1	773.3	739.2	2834.8	6517.7	5226.9
5	1024	8	848.3	792.1	2834.7	3887.5	5341.6
5	1024	64	893.8	935.2	3350.4	3170.2	5923.4

The table is loaded directly from `bench/results/recovery-matched-20260917.json` when the book compiles. That file also records raw samples, source and binary hashes, compiler versions and flags, CPU affinity, and the paired comparison intervals. Use `make bench-matched` to repeat the workloads; see `docs/book/06_measurement_methods.typ` for dependency paths and baseline reconstruction.

10.4.1 Read across workloads before naming a winner

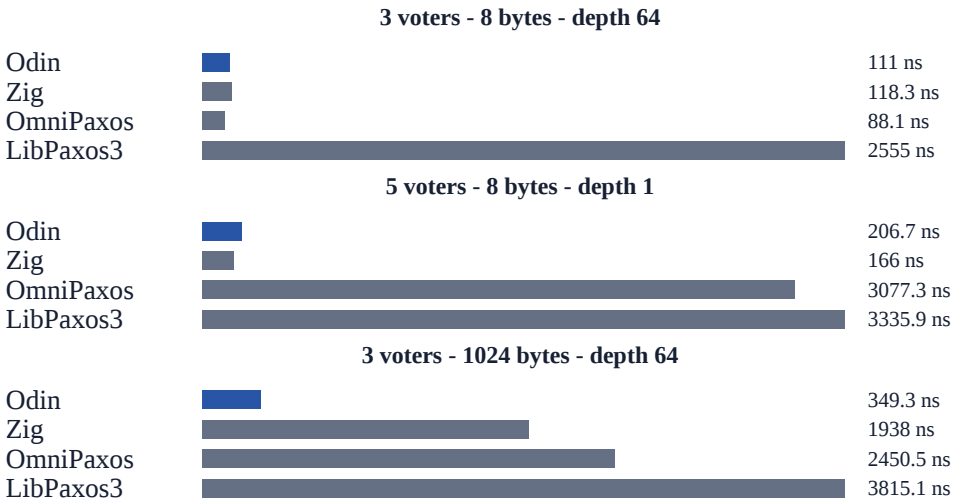


Figure 16: Three slices of the same matrix. Bar length is median time per value and starts at zero. Each panel has its own scale, printed in nanoseconds beside every bar. The best result changes with payload, voter count, and outstanding work.

For three voters and 1 KiB values, the paired median cost decreased by about 16-26% relative to the Odin baseline. Several small-payload workloads became about 1-3% slower. At five voters, 1 KiB, and depth 64, the paired ratio is 1.047 with a 95% bootstrap interval of 0.928-1.089. That interval includes both an improvement and a regression, so this row does not establish either.

The regression gate rejects a workload when its entire paired 95% interval exceeds 1.05. All workloads passed this gate; passing does not prove that every slowdown is smaller than 5%. The five-voter interval above illustrates the distinction. Zig and OmniPaxos lead some categories. These measurements support specific workload claims, not a claim that one library is always fastest.

> **Prediction Exercise**
> Odin is faster at three voters and 1 KiB, but slower than Zig at five voters and 8 bytes. Which row would you use to estimate your service? Name two costs the benchmark leaves out before interpreting the number as request latency.

10.5 Memory: count the storage you mean

Recovery reports are temporary; ledger entries must remain available until the host releases them. Reducing the former from a window to a chunk removes payload storage, metadata, and per-peer bitmap words. For three voters and 1 KiB values:

Window / chunk	Before, bytes	After, bytes	Reduction
256 / 64	633,120	433,176	31.6%
4,096 / 256	9,080,448	5,081,568	44.0%

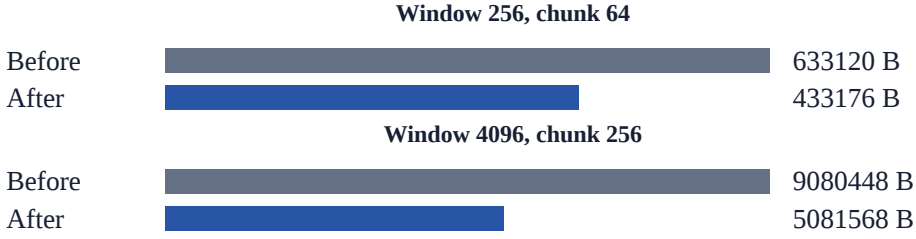


Figure 17: One node plus one effects buffer, three voters and 1 KiB values. Each pair uses its own zero-based scale and prints the byte count. The data comes from the recorded CSV files; the bars exclude queues and application memory.

These totals count one node plus one effects buffer. They exclude transport queues, application state, and runtime overhead. When chunk and window sizes are equal, the node's size is unchanged. The CSV files `recovery-memory-before.csv` and `recovery-memory-after.csv` under `bench/results/` record the configurations.
Three measurements answer different questions:

Measure	Counts	Does not establish
Inline size	Bytes reserved by the configured structs.	Total process memory.
Massif	Instrumented heap and stack allocation.	Static/BSS storage or resident pages.
Sampled peak RSS	Resident process pages observed after execution begins.	Which library field caused the footprint, or every possible peak.

In the recorded three-voter, 1 KiB, depth-64 profile, Odin's sampled peak RSS was 25.4 MB, compared with 52.8 MB for Zig, 32.9 MB for OmniPaxos, and 27.9 MB for LibPaxos3 (decimal MB). Odin did not have the smallest RSS in the small-payload profiles. Massif can exceed RSS when allocated pages are untouched, or miss static storage entirely; never add the two measurements together.

10.6 Profiling an optimisation

A useful profile tests an explanation. The retransmission hypothesis was that a sparse bitmap scan revisited the same occupied cell until it exhausted the chunk budget. A regression test now requires one retry for one used slot. The retained scan wraps at most once and visits each used cell at most once per sweep.
In the dedicated retransmission workload, Callgrind instructions fell from 12,591,988 to 10,139,090 (19.5%). A first attempt that counted occupied cells before scanning used 17,833,184 instructions and was discarded. The retained change's paired native timing ratio was 0.877, with a 95% interval of 0.820-0.905. The steady-state matrix above checks the broader effect; the dedicated result is not a promised speedup for every workload.
`make bench-profile` builds symbolised drivers and collects Callgrind and Massif profiles. Instrumented elapsed time is not a native timing result. Raw traces, annotations, and both retry experiments are archived in `bench/results/recovery-profiles-20260917.tar.gz`; the associated JSON and POD 0009 describe the measurements and their limits.

10.7 Historical durability measurements

The earlier harness also measured a journal and storage barrier. Its CPU rows use a different host path, including a journal replay mirror, so their nanoseconds cannot be compared directly with the matched matrix. The durability table below is retained as evidence from its recorded revision and machine, not as a fresh run of the current source.

With a journal and a storage barrier

AMD Ryzen 7 5800H with Radeon Graphics, Linux 7.0.0-28-generic - odin odin version dev-2026-09-nightly:a2fb372, zig 0.16.0, rustc 1.98.1 (48a229cea 2026-09-01) - Odin build #`bench.meta.odin_build` - recorded 2026-09-16T23:53:16Z

Library	Workload	Mode	Per value	fsync per value
paxos-odin	u64-3n-durable	durable-sync	27.49 ms	6
paxos-odin	u64-3n-durable	durable-pipeline8	3.71 ms	0.75

On that disk, one barrier per host commit round cost tens of milliseconds per value; grouping eight values reduced the cost to a few milliseconds. The lesson is a workload question: if storage dominates the service, a faster consensus transition may make little difference to end-to-end latency. `make bench-compare` runs this historical harness; the matched CPU and profile commands are separate.

10.8 Capability map: exact boundaries

Concern	Core	Boundary
Ballots, promises, votes, commits	yes	`Node` and `Ledger`.
Rotating slot ownership: suggest, skip, revoke, resubmit	yes	`Node_Options.rotating_ownership`; `owner_of` names the proposer of a slot.
Chunked recovery, no-op filling, fences	yes	`start_campaign` through `become_leader`.
Bounded window, memory floor, trim anchors	yes	Host licenses reuse with `advance_memory_floor`; host serves `Serve_Range_Request`.
Stop-sign reconfiguration, configuration-checked envelopes	yes	`Replicated_Log_Node`, `Log_Envelope`.
Non-voting learners	yes	`Learner` and `node_init_learner`.
Runtime durability gate	yes	`Effects` under `.Enforced`; `.Host_Managed` is an audited exception.
Journal format, fsync, replay loop	no	Host: persist `Write` records in order, copying each value out of the ledger; replay with `ledger_replay_fold`.
Transport, codec, authentication	no	Host: `Envelope` in, `Envelope` out; the core trusts `from`.
Client sessions and deduplication	no	Host state machine; see the key-value design in Part V.
Snapshot store and state images	no	Host; the core only carries `Trim_Anchor`.
Linearizable reads, leases	no	Not implemented. `is_leader_caught_up` reports prefix progress only. Leases are a proposal (POD 0004).
Byzantine tolerance	no	Out of scope by design.

10.9 Operating drills with exit criteria

Every drill below can be run against the simulator today and against a real deployment once a host exists. A drill without an exit criterion is a demonstration, not a test.

Drill	Procedure	Passes when
Follower crash	Kill one follower mid-stream; keep proposing.	Throughput continues; on restart the follower's `decided_through` reaches the leader's within one resend interval.
Leader crash during a vote	Kill the leader after `Write_Vote` is durable and before `Commit` leaves.	A new leader is elected; the vote's value is chosen, never a different one; the client that timed out sees its request applied exactly once after retry.
Minority partition	Isolate fewer than a read quorum of voters.	The majority keeps deciding; the minority's leader, if any, steps down on the first `Nack`; on healing the minority catches up without a divergent slot.
Disk full or sync failure	Make the journal append or `fsync` fail on one node.	The host never calls `confirm_writes_durable` for that batch; it stops the node and restarts

Drill	Procedure	Passes when
		from the journal; no message from the failed batch was sent.
Corrupt state image	Install a state image whose anchor does not match the certified trim.	`begin_recovery` or `install_chosen_trim` returns `Trim_Regression`; the node does not resume.
Window backpressure	Stop applying on one node while the leader keeps proposing.	`propose` returns `Window_Full` at the leader once the unapplied prefix reaches `WINDOW_SLOTS`; it resumes when the floor advances.
Crashed owner	Under rotating ownership, kill one owner while the others keep proposing.	After `election_timeout_ticks` of stall a survivor revokes the stalled chunk; the log advances with no-ops in the dead owner's slots; on restart the owner resumes in its own slots above the revoked range.

Exercise 19.1
Add an oracle to the simulator that rejects a `Commit_Message` whose value differs from a durable write-quorum decision for the same slot, even when the sender is not the leader. Say which existing oracle already implies it and why the new one is still worth its cost.

Review & Discussion
Compare the complementary roles of testing and formal reasoning:

- Why a green unit test suite only verifies pre-selected execution traces.
- How pseudo-random fault injection explores vast schedule permutations against continuous invariant oracles.
- Why even extensive empirical simulation cannot substitute for the mathematical safety argument established in Part III.

11 Reproducing Measurements

The drivers in `bench/matched/` measure the libraries without adding adapters to the consensus core. Run the commands below from the repository root. The library has no dependency on C, Rust, Valgrind, network services, storage, or threads.

11.1 Run

Set `ZIG`, `ODIN`, and `CARGO` when the compilers are not on PATH. The Zig checkout is selected with `PAXOS_ZIG_DIR` (default `../paxos-zig`). LibPaxos must already be available at `LIBPAXOS_SOURCE`, or the sibling benchmark cache. Its revision is checked against `d255f8b67a32d5e0ef43ac1a393b72cee23d8e0e`. OmniPaxos is pinned to 0.2.2 with a checked-in Cargo lockfile. No upstream sources are edited.

```
python3 tools/bench_compare.py --matched --smoke
python3 tools/matched_compare.py --baseline-root=/path/to/preserved/odin/tree
python3 tools/matched_compare.py --profile-build --build-only --output=bin/profile-build.json
python3 tools/matched_profile.py bin/profile-build.json
python3 tools/profile_recovery.py --baseline-root=/path/to/preserved/odin/tree
python3 tools/test_matched_tools.py
```

`--only=3,1024,64` selects members, payload bytes, and outstanding commands. `--implementations=odin,zig` restricts a run. Defaults cover all four implementations, three/five voters, 8/64/1024-byte payloads, and depths 1/8/64. Profiles use three representative rows; recovery, moving windows, and retransmission have separate Odin profiles. Unavailable tools fail with diagnostics; missing results are never reported as zero cost.

11.2 Workload contract

Each measured epoch starts with an elected stable leader and submits 4,096 unique commands. A separate epoch warms the process. Each outstanding group is delivered until its message queue is empty. Decision completion is timed; full ordered payload validation at every configured learner follows outside the timing interval. Payloads are fixed arrays of 64-bit words with the sequence number in the first word and zero padding. Drivers validate all words, not merely checksum sums.

There is no serialization, I/O, journal replay mirror, or network delay. Odin/Zig have window 4,096, chunk 256, and a caller-owned effects buffer. Their durable-write barrier is a no-op in this cost model. Rust uses the upstream in-memory backend. LibPaxos uses its native memory backend and one learner per voting member; its phase-one preexecution remains timed. C's delivered-value buffer belongs to the benchmark host and is included in process memory measurements.

The pipeline depth bounds outstanding client commands, not protocol message count. OmniPaxos may coalesce them. LibPaxos's proposer and one learner are co-located when counting accepted-message deliveries. These native differences are reported, not removed by rewriting library behavior. No cross-language source translation is used in the Odin library. Rust driver setup/delivery helpers are adapted from the MIT-licensed paxos-zig benchmark; the other matched drivers are local harnesses.

This is a finite-retention comparison. It does not claim identical long-running trim behavior or feature coverage. API batching, ownership, and durable modes are explicitly outside the common matrix; the existing Odin benchmark retains those additional scenarios separately. Ownership settling in the historical benchmark is outside its timed interval, unlike decision completion in this common matrix.

11.3 Timing and evidence

The runner pins one allowed CPU where possible, records container CPU/memory limits, rotates execution order, and collects nine samples. A common epoch count is chosen from pilots, targeting 20 ms for the fastest implementation, capped at 16 epochs. This cap bounds slow-driver cost; raw samples reveal remaining timer/noise effects. Compilation and warm-up are excluded. Production builds use each language's optimized native target; profile builds retain symbols and use portable targets.

Results retain raw samples, medians, quartiles, message counts, source and binary hashes, dependency versions, commands, and static components where available. A paired deterministic bootstrap estimates candidate/baseline ratios. The regression gate fails when its 95% interval is entirely above a 5% slowdown. Smoke measurements are report-only and never support performance claims. Existing results are not overwritten unless the caller explicitly supplies their path.

Do not run a timing comparison concurrently with builds, tests, or instrumented profiles. Different host/toolchain/build-policy measurements are not attributable solely to source changes. A baseline tree must be preserved before editing; the runner uses the same current driver against both library trees.

11.4 CPU and memory profiles

Callgrind collection is toggled only inside `measured_epoch`, excluding warm-up, setup, and validation. Its cache and branch results are simulations, not hardware counters. Instrumented elapsed times are never included in comparison rows.

Massif reports allocated heap, allocation overhead, and stack use. It does not count static/BSS storage, which is substantial in Zig's fixed-capacity driver. Post-exec `/proc` high-water RSS is sampled separately over eight epochs; it is null if the child exits before observation. This avoids counting Python's inherited pre-exec RSS. RSS includes runtime/libraries and resident static storage, and can be lower than reserved heap capacity because untouched pages need not be resident.

`tools/memory_report.odin` reports core Odin sizes independently. Node already includes its ledger; do not add the two. Do not add static capacity, Massif, and RSS together: they overlap and describe different aspects of memory use.

11.5 Reconstructing the recorded baseline

`bench/results/recovery-baseline.json` identifies a repository commit and a patch that reconstruct the exact pre-refactor `src` tree. Apply that patch in an isolated checkout of the recorded commit, then pass that checkout as `--baseline-root`. The reconstruction was checked against its recorded source SHA-256. The benchmark uses the current matched driver for both source trees; older timing harnesses are not mixed into the before/after comparison.

11.6 Measuring the static memory budget

Run from the repository root:

```
odin run tools/memory_report.odin -file -out:/tmp/paxos-memory-report
```

The CSV reports actual instantiated type sizes. `node_bytes` already includes `ledger_bytes`; `total_bytes` adds one caller-owned effects buffer to one node. It excludes transport queues, serialization buffers, journals, allocator overhead, application state, and process/runtime memory. A host may share one effects buffer between nodes if it consumes or copies each batch before reuse.

On x86-64 (Odin dev-2026-09-nightly:a2fb372), after chunk-local recovery:

Bytes / value	Voters	Window	Chunk	Node bytes	Effects bytes	Total bytes
8	3	256	64	18296	22704	41000
1024	3	256	64	410472	22704	433176
8	5	256	64	18464	32272	50736
1024	5	256	64	410640	32272	442912
8	3	256	256	24704	76464	101168
1024	3	256	256	807024	76464	883488
8	3	4096	256	259904	137904	397808
1024	3	4096	256	4943664	137904	5081568

For three members and 1 KiB values, the previous node+effects sizes were 633,120 bytes at window 256/chunk 64 and 9,080,448 bytes at window 4,096/chunk 256. The reductions are 199,944 bytes (31.6%) and 3,998,880 bytes (44.0%) respectively.
Borrowed payloads keep effects storage independent of payload size. The node now has one window of ledger values, one **chunk** of recovery values, and a chunk-sized resubmission queue. Recovery metadata and per-peer duplicate bitmaps also scale with the chunk. Selection-freeze flags occupy existing alignment space rather than adding another allocation. Public procedures and journal/wire formats are unchanged; consumers must recompile for the new internal Node layout.
Canonical membership removes the separate member index. Duplicate validation scans adjacent sorted IDs rather than comparing every pair of input IDs.

11.6.1 Comparing implementations

Match member count, window, recovery chunk, payload, durability policy, and enabled features. Measure transport queues and peak resident memory separately from these static sizes. The matched suite reports Odin/Zig inline storage and measures all four processes with Massif and sampled post-exec RSS; see the workload contract above.
The matched measurements in POD 0009 show workload-dependent advantages. Odin leads the recorded large-payload cases; Zig and OmniPaxos lead some small-payload cases. Static sizes do not establish a universal memory winner: the report includes heap/stack profiles and sampled resident memory separately.
The older ownership harness settles gaps outside its timed interval, so those rows are not measurements of end-to-end completion latency. Historical files remain attributable to their recorded source and harness; the matched results are archived separately in `bench/results/recovery-matched-20260917.json`.

11.7 Measuring the Python SDK across its four paths

The native benchmark measures neither Python object creation nor a Python journal, so the SDK is measured on its own terms: the same workload driven through native Odin, the raw C ABI, the typed `Node` and the durable `Session`, with membership, payload, capacities and completion rule held equal.

```
python3 tools/paxodin_measure.py --iterations 10000 --samples 9
python3 tools/paxodin_measure.py --smoke          # one sample, for a quick check
```

It builds `tools/paxodin_native_bench.odin` with the same flags the benchmark uses, then drives each Python path through `tools/paxodin_paths.py`, and writes `bench/results/paxodin-paths-<date>.json`. That file records the CPU, the Odin and Python versions, the core revision, whether the tree was dirty, a SHA-256 over every source that took part, the exact commands, and every raw sample. Ratios are paired bootstrap medians with a 95% interval.
Two rules govern reading the result. **Transition-only work and durable host work are separate measurements**: the `native`, `abi` and `node` paths touch no storage and no network, `session_memory` adds framing and an in-memory journal, and `session_durable` adds an `fsync` per batch. Comparing across that boundary - a Python `fsync` against a native in-memory transition - measures nothing. And **the counts matter more than the times**: each row reports boundary crossings, bytes copied across the ABI and sync calls per value, which is what tells you whether a batched binding or a compiled extension would address the cost at all.
Run it with nothing else on the machine. A concurrent Typst compile or test run has visibly skewed this repository's benchmarks before.

PART VII

Desk reference

Messages, effects, the public API, options, errors, formulas, invariants, exercise answers, and sources, collected where a reviewer can reach them without rereading the narrative chapters.

12 Consensus Desk Reference

This chapter is the part of the book meant to be opened rather than read. Every identifier is spelled as it appears under `src/`, with the parameter order the compiler accepts. Where the prose and the code disagree, the code is right and this page needs a patch. `VERSION` is `"0.2.0"`.

12.1 Message reference

`Message(Value)` is a union of nine variants. `Envelope(Value)` wraps one with `from` and `to`, both `Node_Id` (a `u16`; zero is reserved). `node_step` answers `.Wrong_Recipient` when `to` is not the local id and `.Not_Member` when `from` lies outside the membership; a non-voting node accepts only `Commit_Message` and answers every other kind with `.Learner_Message_Forbidden`. Three variants carry a value as a `^Value`: outbound it points into the sender's ledger and is valid until that node's next transition; inbound the host points it at the decoded value for the duration of `step`. `message_value(message) -> (^Value, bool)` returns that pointer for the kinds that carry one (a `Promise_Message` only when its `state` is not `.Empty`).

Variant	Fields	Meaning and sender
`Prepare_Message`	`ballot: Ballot`, `first: Slot`, `last: Slot`, `scope: Prepare_Scope`	Phase one. `start_campaign` and `begin_next_chunk` broadcast it to every member, the sender included, with `scope = .Global`: promise `ballot` for every decree from `first` on and report votes in `[first, last]`. `start_revocation` sends it with `scope = .Bounded`: promise only the decrees in `[first, last]`, recorded per decree.
`Promise_Message(Value)`	`ballot: Ballot`, `slot: Slot`, `vote: Ballot`, `state: Cell_State`, `value: ^Value`	One reported vote (`state = .Voted`) or decision (`state = .Chosen`) for one decree in the chunk. `on_prepare` sends one per used cell above the trim anchor and at or above `first`; a reported decision dominates every vote in `on_promise`.
`Promise_Range_Message`	`ballot: Ballot`, `anchor: Trim_Anchor`, `chosen_through: Slot`, `first: Slot`, `last: Slot`, `reported: u32`, `more: bool`	The manifest that closes a phase-one answer. `reported` tells the candidate how many `Promise_Messages` describe `[first, last]`; `more` says used cells lie above `last`; `chosen_through` is the acceptor's released prefix; `anchor` is its adopted trim record.
`Accept_Message(Value)`	`ballot: Ballot`, `slot: Slot`, `value: ^Value`	Phase two. `send_accept` records the sender's own vote, then broadcasts to every peer; `resend_to` repeats it. It is the only variant `pre_durable_next` yields, and only when `ballot_round(ballot) > 0`.
`Accepted_Message`	`ballot: Ballot`, `slot: Slot`, `decided_through: Slot`	`on_accept` answers after recording `Write_Vote`; a duplicate of an identical vote is acknowledged again without a new write. `decided_through` updates the sender's entry in `peer_decided_through`.

Variant	Fields	Meaning and sender
`Commit_Message(Value)`	`slot: Slot`, `value: ^Value`	A value the sender knows is chosen: `on_accepted` broadcasts it once a write quorum acknowledged; `resolve_chunk`, `on_learn`, and `resend_to` re-teach it; `on_accept` answers a conflicting accept for a decided cell with it; the host serves evicted history with it.
`Learn_Message`	`from_slot: Slot`, `count: u32`	Catch-up request for decisions in `from_slot` through `from_slot + count - 1`, `count` in `1..=CHUNK_SLOTS`. Sent by `request_learn` from `on_heartbeat` when behind, from `resolve_chunk` toward the peer with the longest chosen prefix, from `resend_to`, `node_reconnected`, `tick_ownership`, and `node_request_catch_up`.
`Nack_Message`	`rejected: Ballot`, `promised: Ballot`, `slot: Slot`, `decided_through: Slot`	Refusal of a ballot below the promise, from `on_prepare` or `on_heartbeat` (`slot = 0`) or `on_accept` (the refused decree). `on_nack` steps the rejected candidate or leader down and records `ballot_node(promised)` as the leader hint.
`Heartbeat_Message`	`ballot: Ballot`, `decided_through: Slot`	Leader liveness, broadcast by `node_tick` every `heartbeat_interval_ticks`. A follower whose promise is lower adopts the ballot with a `Write_Promise`; one that is behind replies with `Learn_Message`.

`Prepare_Scope` is `enum u8 { Global, Bounded }`. `Cell_State` is `enum u8 { Empty, Voted, Chosen }`.

12.2 Effect ordering

An `Effects` batch is the whole output of one transition. It holds four lists and one flag, `writes_pending`, which `effects_add_write` raises and `effects_confirm_writes_durable` lowers. `Write(Value)` is a union of five records:

Write	Fields	Must be durable before
`Write_Promise`	`ballot: Ballot`	Any `Promise_Message` or `Promise_Range_Message` for that ballot leaves (`on_prepare`, `scope = .Global`), and any `Nack_Message` that cites it as `promised`. `on_heartbeat` also emits it.
`Write_Promise_At`	`ballot: Ballot`, `slot: Slot`	The bounded answer leaves: `promise_bounded` emits one per decree in `[first, last]` above the memory floor whose per-decree promise changes.
`Write_Vote(Value)`	`ballot: Ballot`, `slot: Slot`, `value: ^Value`	The `Accepted_Message` for that vote leaves (`on_accept`), and before the leader counts its own vote toward a decision (`send_accept`).

Write	Fields	Must be durable before
`Write_Chosen(Value)`	`slot: Slot`, `value: ^Value`	The `Commit_Message` broadcast, the application's consumption of the matching `Committed` entry, and any `on_learn` answer built on it (`record_commit`).
`Write_Trim`	`Trim_Anchor` (`trim_id: u64`, `chosen_trim_slot: Slot`)	Any `Promise_Range_Message` that vouches for the released prefix from this anchor, and any cell reuse below it (`node_install_chosen_trim`).

Pointer validity

Nothing in a batch owns a value. `Write_Vote.value`, `Write_Chosen.value`, every `^Value` in a message, and `Committed.value` point into the ledger of the node that produced the batch (or, for one decision released past the window edge, into `node.pass_through`). Each pointer is valid until the next transition on that node. A host persists every write before it runs another transition, and copies or serialises a message's value when it queues the envelope, as `packet_of` does in `examples/counter.odin`.

`Committed(Value)` carries `slot` and `value: ^Value`; entries arrive in slot order, contiguous with everything released earlier (`emit_contiguous`). `Host_Request` is a union with one variant, `Serve_Range_Request`, carrying `peer: Node_Id`, `first: Slot`, and `count: u32`: `on_learn` emits it when a peer asks for history at or below `memory_floor`, and the host answers from its own journal or state image with `Commit_Messages`.

Host commit sequence

1. `effects_writes_slice`: append every record in order and sync.
2. `effects_confirm_writes_durable`: clear `writes_pending`. Never confirm a failed write; recover from the journal instead.
3. `effects_committed_slice`: apply released entries in order.
4. `effects_requests_slice`: serve history the peer asked for.
5. `effects_messages_slice`: transmit. Under the default `Durability_Gate.Enforced`, reading this slice while writes are pending stops the process, and so does `effects_reset` on an unconfirmed batch.

Every public transition calls `effects_reset` first, so a host drains one batch fully before the next call. `effects_init` discards without the check; it exists for fresh memory and for a crashed batch that can never be completed. `.Host_Managed` disables the check for a host that has been audited against the four rules in `src/effects.odin`.

`effects_requires_power_loss_barrier` returns true when the batch holds a `Write_Promise`, `Write_Promise_At`, or `Write_Vote`. Decision and trim records are derived facts that a host may persist with a cheaper barrier. `effects_pre_durable_messages` returns a `Pre_Durable_Iterator(Value)`; `pre_durable_next` yields only `Accept_Message` envelopes whose `ballot_round` is above zero. A campaign accept may leave before the local sync because it claims nothing about the sender's durable state and a restarted proposer campaigns at a fresh ballot; an owner's round-zero suggestion must wait, because its own vote is the only durable record that the instance was used. `effects_is_empty` reports a transition that produced nothing to do.

12.3 Core API

In the signatures below `node` is `^Node($V, $M, $W, $C, $G)` and `effects` is the matching `^Effects(V, M, W, C, G)`; `m` is `^Membership($MAX_MEMBERS)`; `l` is `^Ledger($Value, $WINDOW)`; `e` is `^Effects($V, $M, $W, $C, $G)`; `bs` is a `Bit_Set($N)`, by pointer where it is mutated. Transitions reset `effects` before doing anything else.

12.3.1 Membership

Proc	Contract
`membership_init(m, node_ids: []Node_Id, read_quorum_override: int = 0, write_quorum_override: int = 0) -> Error`	Validate non-zero unique ids and quorum sizes; zero overrides mean majority. Errors leave `m` untouched.
`membership_index_of(m, id: Node_Id) -> (int, bool)`	The stable index of `id` (its rank, since members are sorted). Linear scan up to `LINEAR_LOOKUP_LIMIT` members, binary search above it.
`membership_contains(m, id: Node_Id) -> bool`	Membership test.
`membership_count(m) -> int`	Number of voters.
`membership_get(m, index: int) -> Node_Id`	The member at a stable index.
`membership_slice(m) -> []Node_Id`	Members in ascending id order.
`membership_read_quorum(m) -> int`, `membership_write_quorum(m) -> int`	Phase-one and phase-two quorum sizes.

`Membership(MAX_MEMBERS)` holds `members` (a `Small_Array` sorted by id whatever order the host listed them in; the position is the member's stable index, and under rotating ownership the owner order), `read_quorum_size`, and `write_quorum_size`.

12.3.2 Ballots, slots, and bit sets

Proc	Contract
`ballot_make(round: u64, priority: u8, node: Node_Id) -> Ballot`	Pack `round << 24 | priority << 16 | node` into one `distinct u64`, so `<` on `Ballot` orders round, then priority, then node.
`ballot_round(b: Ballot) -> u64`, `ballot_priority(b: Ballot) -> u8`, `ballot_node(b: Ballot) -> Node_Id`	Unpack the three fields.
`cell_of(slot: Slot, $WINDOW: int) -> int`	The window cell of a slot: `(slot - 1) & (WINDOW - 1)`.
`slot_add(slot, offset: Slot) -> Slot`	Add without wrapping past `max(Slot)`.
`bit_set_insert(bs, index: int) -> bool`	Insert; true when the index was absent.
`bit_set_remove(bs, index: int)`, `bit_set_contains(bs, index: int) -> bool`, `bit_set_count(bs) -> int`, `bit_set_reset(bs)`	Remove, test, cardinality, clear.
`bit_set_next(bs, from: int) -> (int, bool)`	The smallest member at or after `from`, by trailing-zero count per 64-bit word.
`bit_set_last(bs) -> (int, bool)`	The largest member.

`Bit_Set(N)` is `[(N + WORD_BITS - 1) / WORD_BITS]Word` with `Word :: bit_set[0..<WORD_BITS]` and `WORD_BITS :: 64`. `Ballot` constants: `BALLOT_ZERO`, `BALLOT_ROUND_BITS` (40), and `MAX_ROUND` ($2^{40} - 1$).

12.3.3 Ledger

Proc	Contract
`ledger_promise_for(l, cell: int) -> Ballot`	The effective promise for a cell: `max(promised, promised_at[cell])`.

Proc	Contract
`ledger_cell(l, slot: Slot) -> (int, bool)`	The cell and whether it currently holds `slot`.
`ledger_vote_at(l, slot: Slot) -> (Ballot, ^Value, bool)`	The vote in a `.Voted` cell for `slot`.
`ledger_chosen_at(l, slot: Slot) -> (^Value, bool)`	The decision in a `.Chosen` cell for `slot`.
`ledger_is_chosen(l, slot: Slot) -> bool`	Decision test.
`ledger_open(l, cell: int, slot: Slot)`, `ledger_clear_cell(l, cell: int)`	Retag a cell for `slot` (or for no slot), clearing everything but the value storage.
`ledger_claim(l, slot: Slot) -> (int, bool)`	Replay-time claim: reuse a cell only when its previous slot is chosen or at or below the anchor.
`ledger_record_vote(l, cell: int, ballot: Ballot, value: Value)`, `ledger_record_chosen(l, cell: int, value: Value)`	Store a vote or a decision and maintain the `used` and `chosen` bitmaps.
`ledger_highest_ballot(l) -> Ballot`	The greatest of the global promise, every per-decree promise, and every vote.
`ledger_highest_used(l) -> Slot`	The greatest slot held by any used cell.
`ledger_apply(l, write: Write(Value)) -> Error`	Strict single-configuration replay of one record (rules in Part VIII).
`ledger_replay_fold(l, write: Write(Value)) -> Error`	Lifetime replay across window reuse: promises fold to their maximum, a vote or per-decree promise whose cell a later slot owns is skipped, decisions and anchors stay strict.

`Ledger(Value, WINDOW)` is struct-of-arrays: `promised: Ballot`, `anchor: Trim_Anchor`, and per cell `slot`, `promised_at`, `vote_ballot`, `state: Cell_State`, and `value`, plus the bitmaps `used` and `chosen`. The host never copies it; it persists the five `Write` records and rebuilds the struct by replay.

12.3.4 Node lifecycle and queries

Proc	Contract
`node_init(node, id: Node_Id, membership: Membership(M), options := Node_Options{}) -> Error`	Voting follower at slot 1; `.Not_Member` if `id` is not a voter.
`node_init_learner(node, id: Node_Id, membership: Membership(M)) -> Error`	Non-voting learner; an `id` inside the membership is `.Learner_Is_Voter`.
`node_restore(node, id: Node_Id, membership: Membership(M), ledger: Ledger(V, W), floor: Slot = 0, options := Node_Options{}) -> Error`	Rebuild from a replayed `Ledger`; open votes at or below `max(floor, anchor)` are dropped.
`node_continue_at(node, id: Node_Id, membership: Membership(M), floor: Slot, anchor: Trim_Anchor, options := Node_Options{}) -> Error`	Empty node resuming at `floor + 1` with an inherited anchor.
`node_restore_learner(node, id: Node_Id, membership: Membership(M), ledger: Ledger(V, W)) -> Error`	Learner from its decision-only journal.

Proc	Contract
`node_begin_recovery(node, anchor: Trim_Anchor) -> Error`	Install a certified prefix; keeps cells above the anchor, returns to `.Follower`, persists nothing.
`node_advance_memory_floor(node, through: Slot) -> Error`	Host has durably consumed the released prefix; `.Invalid_Slot` above `delivered_through`.
`node_install_chosen_trim(node, anchor: Trim_Anchor, effects) -> Error`	Adopt a chosen trim record, emitting `Write_Trim`.
`node_set_campaign_enabled(node, enabled: bool)`, `node_is_campaign_enabled(node) -> bool`	Toggle elections; disabling during `.Preparing` drops back to `.Follower`.
`node_current_leader(node) -> (Node_Id, bool)`	The leader hint, if any.
`node_decided_through(node) -> Slot`	Greatest contiguous slot released.
`node_leader_base(node) -> Slot`	First slot the current leadership may fill.
`node_proposal_frontier(node) -> Slot`	Slot the next proposal would take.
`node_is_leader_caught_up(node) -> bool`	`delivered_through >= leader_base - 1`; not a lease and not a read barrier.
`node_memory_floor(node) -> Slot`, `node_trim_anchor(node) -> Trim_Anchor`	Reuse floor and adopted anchor.
`node_role(node) -> Role`, `node_ballot(node) -> Ballot`, `node_id(node) -> Node_Id`, `node_is_voting_member(node) -> bool`	Plain accessors. `Role` is `enum u8 { Follower, Preparing, Leader }`.
`node_ledger(node) -> ^Ledger(V, W)`	Inspection only; persistence goes through `Write` records.
`node_resubmits_dropped(node) -> u32`	Losing suggestions that could not be queued for resubmission (the queue holds one chunk). Resubmission is best effort; the host retries these.
`node_committed_at(node, slot: Slot) -> (V, bool)`	A resident decided value, by copy.
`node_read_decided(node, from_slot: Slot, output: []Committed(V)) -> (int, Error)`	Copy the released suffix; `.Trimmed` at or below the floor, `.Read_Buffer_Too_Small`.

12.3.5 Node transitions

Proc	Contract
`node_campaign(node, noop: V, effects) -> Error`	Start phase one; `.Not_Voter`, or `.Campaign_Disabled` when disabled or under rotating ownership.
`node_propose(node, value: V, effects) -> (slot: Slot, err: Error)`	Assign the next slot (or the next own slot) and send accepts; `.Not_Leader`, `.Leader_Catching_Up`, `.Window_Full`.
`node_propose_batch(node, values: []V, slots: []Slot, effects) -> (assigned: []Slot, err: Error)`	Up to `CHUNK_SLOTS` values into consecutive slots (consecutive own slots under ownership), all or none.

Proc	Contract
`node_tick(node, noop: V, effects) -> Error`	Advance the election, heartbeat, and resend timers by one; under ownership, skips, resubmissions, and stall detection.
`node_step(node, envelope: Envelope(V), effects) -> Error`	Process one authenticated envelope addressed to this node.
`node_learn_chosen(node, from: Node_Id, slot: Slot, value: V, effects) -> Error`	Learner-only: install a host-certified decision; a voter gets `.Not_Learner`.
`node_reconnected(node, peer: Node_Id, effects) -> Error`	Repair one peer path: a leader resends, a follower asks its leader to teach.
`node_request_catch_up(node, peer: Node_Id, from_slot: Slot, effects) -> Error`	Emit one chunk-bounded `Learn_Message`.
`owner_of(node, slot: Slot) -> Node_Id`	The member that owns `slot`: `membership_get(m, (slot - 1) mod N)`.
`ownership_ballot(owner: Node_Id) -> Ballot`	`ballot_make(0, 0, owner)`, the round-zero ballot no campaign ever uses.

12.3.6 Effects

Proc	Contract
`effects_init(e)`	Discard without checking the gate (fresh memory, crashed batch).
`effects_reset(e)`	Empty for the next transition; stops the process on unconfirmed writes under `.Enforced`.
`effects_confirm_writes_durable(e)`	Every write is appended and synced.
`effects_writes_slice(e) -> []Write(V)`	Durable records, in order.
`effects_messages_slice(e) -> []Envelope(V)`	Outbound envelopes; stops the process while writes are pending under `.Enforced`.
`effects_committed_slice(e) -> []Committed(V)`	Newly decided entries, contiguous.
`effects_requests_slice(e) -> []Host_Request`	History a peer asked for.
`effects_requires_power_loss_barrier(e) -> bool`	The batch holds a promise or a vote.
`effects_pre_durable_messages(e) -> Pre_Durable_Iterator(V)`, `pre_durable_next(it: ^Pre_Durable_Iterator($Value)) -> (Envelope(Value), bool)`	The accept-only iterator for campaign ballots.
`effects_is_empty(e) -> bool`	All four lists are empty.
`effects_add_write(e, w: Write(V))`, `effects_add_message(e, envelope: Envelope(V))`, `effects_add_committed(e, c: Committed(V))`, `effects_add_request(e, r: Host_Request)`	Producers used by the node; a full list is an assertion failure, never backpressure.

12.3.7 Proc groups

`src/paxos.odin` folds these into proc groups dispatched on the receiver type: `init`, `init_learner`, `restore`, `restore_learner`, `continue_at`, `begin_recovery`, `campaign`, `propose`, `propose_batch`, `step`, `tick`, `reconnected`, `request_catch_up`, `learn_chosen`, `set_campaign_enabled`, `advance_memory_floor`, `install_chosen_trim`, `current_leader`, `decided_through`, `leader_base`, `proposal_frontier`, `committed_at`, `read_decided`, `is_leader_caught_up`, `is_campaign_enabled`, `memory_floor`, `trim_anchor`, `role`, `ballot`, `id`, `is_voting_member`, `resubmits_dropped`, and `ledger`. `init` also covers `effects_init`, `membership_init`, and `stop_sign_init`; `step` covers both replicated-log overloads; `committed_at` covers `replicated_log_read` and `learner_chosen_at`; `read_decided` covers `learner_read_chosen`; `learn_chosen` covers `learner_learn_chosen`. The effects verbs drop their prefix: `reset`, `confirm_writes_durable`, `writes_slice`, `messages_slice`, `committed_slice`, `requests_slice`, `requires_power_loss_barrier`, `pre_durable_messages`, `is_empty`. The long spellings remain available when a call site wants to name its receiver.

12.4 Replicated-log API

`Replicated_Log_Node(Value, MAX_MEMBERS, WINDOW_SLOTS, CHUNK_SLOTS, MAX_METADATA_BYTES, GATE)` wraps a core node whose value type is `Entry(Value, MAX_MEMBERS, MAX_METADATA_BYTES)`, a union of `Value` and `Stop_Sign(MAX_MEMBERS, MAX_METADATA_BYTES)`. The matching effects type is `Effects(Entry(...), MAX_MEMBERS, WINDOW_SLOTS, CHUNK_SLOTS, GATE)`. A `Stop_Sign` holds `configuration_id`, a `members` small array, and opaque `metadata` bytes; `stop_sign_init(ss, configuration_id, members, metadata)` and `stop_sign_create(T, configuration_id, members, metadata)` validate it, `stop_sign_validate_members(members, $MAX_MEMBERS)` checks a bare member slice, and `stop_sign_members_slice` and `stop_sign_metadata_slice` read it back. `Log_Envelope(Value, MAX_MEMBERS, MAX_METADATA_BYTES)` pairs a core envelope with the `configuration_id` it was sent under.

Proc (short spelling)	Contract
`replicated_log_init(node, id, configuration_id, membership, options := Node_Options{}) -> Error` (`log_init`)	Voting member of a non-zero configuration.
`replicated_log_init_learner(node, id, configuration_id, membership) -> Error` (`log_init_learner`)	Non-voting learner of that configuration.
`replicated_log_init_from_stop(node, id, stop, stop_slot, anchor, options := Node_Options{}) -> Error` (`log_init_from_stop`)	Start the configuration a decided stop sign names, at `stop_slot + 1`; a removed voter gets `.Not_Member`.
`replicated_log_continue_at(node, id, configuration_id, membership, floor, anchor, options := Node_Options{}) -> Error` (`log_continue_at`)	Empty node resuming at `floor + 1`.
`replicated_log_restore(node, id, configuration_id, membership, ledger, floor := 0, options := Node_Options{}) -> Error` (`log_restore`)	Replayed restore; rediscovers a decided seal from the chosen cells.
`replicated_log_restore_learner(node, id, configuration_id, membership, ledger) -> Error` (`log_restore_learner`)	Learner restore.
`replicated_log_begin_recovery(node, anchor) -> Error` (`log_begin_recovery`)	Install a state image; forgets a decided seal and rediscovers it from what remains.

Proc (short spelling)	Contract
`replicated_log_propose(node, value, effects) -> (Slot, Error)` (`log_propose`)	Propose a command; `.Log_Sealed` once a stop is pending or decided.
`replicated_log_propose_batch(node, values, slots, effects) -> ([]Slot, Error)` (`log_propose_batch`)	At most `CHUNK_SLOTS` commands in one batch, else `.Batch_Too_Large`.
`replicated_log_propose_stop_sign(node, next_configuration_id, next_members, metadata, effects) -> (Slot, Error)` (`log_propose_stop_sign`, `log_reconfigure`)	Propose the seal; a next id that is not strictly greater is `.Configuration_Id_Regression`.
`replicated_log_campaign(node, noop, effects) -> Error`, `replicated_log_tick(node, noop, effects) -> Error` (`log_campaign`, `log_tick`)	Core transitions with the `Value` no-op wrapped as an `Entry`.
`replicated_log_step(node, envelope, effects) -> Error`, `replicated_log_step_checked(node, message, effects) -> Error` (`log_step`)	Bare envelope, or a `Log_Envelope` whose id must match; a mismatch is `.Configuration_Mismatch` with no effects.
`replicated_log_envelope(node, envelope) -> Log_Envelope` (`log_envelope`)	Stamp an outbound envelope with the local configuration id.
`replicated_log_learn_chosen(node, from, slot, entry, effects) -> Error` (`log_learn_chosen`)	Learner-only certified decision.
`replicated_log_reconnected`, `replicated_log_request_catch_up`, `replicated_log_advance_memory_floor`, `replicated_log_install_chosen_trim`, `replicated_log_set_campaign_enabled`	Pass-throughs to the core (`log_reconnected`, `log_request_catch_up`, `log_advance_memory_floor`, `log_install_chosen_trim`).
`replicated_log_is_sealed(node) -> bool` (`log_is_sealed`)	A stop sign is pending or decided.
`replicated_log_stop_sign(node) -> (Stop_Sign, bool)` (`log_stop_sign`, `log_is_reconfigured`)	The decided seal, which alone licenses handover.
`replicated_log_stop_slot(node) -> Slot` (`log_stop_slot`)	Its slot, or zero.
`replicated_log_pending_stop_sign(node) -> (Stop_Sign, bool)` (`log_pending_stop_sign`)	A decided seal first, else one retained in any used cell.
`replicated_log_read(node, slot) -> (Entry, bool)`, `replicated_log_read_decided(node, from_slot, output) -> (int, Error)` (`log_read`, `log_read_decided`)	Resident decided entries.
`replicated_log_decided_through`, `replicated_log_leader_base`,	Frontier queries (`log_decided_through`, `log_leader_base`,

Proc (short spelling)	Contract
`replicated_log_proposal_frontier`, `replicated_log_memory_floor`, `replicated_log_trim_anchor`, `replicated_log_current_leader`, `replicated_log_is_leader_caught_up`	`log_proposal_frontier`, `log_memory_floor`, `log_trim_anchor`, `log_current_leader`).
`replicated_log_configuration_id(node) -> u64` (`log_configuration_id`)	The host-supplied configuration identity.
`replicated_log_role`, `replicated_log_ballot`, `replicated_log_id`, `replicated_log_is_voting_member`, `replicated_log_is_campaign_enabled`	Accessors mirroring the core.
`replicated_log_ledger(node) -> ^Ledger(Entry(...), WINDOW_SLOTS)`	The core ledger, for inspection.

12.5 Learner API

`Learner(Value, MAX_ENTRIES)` is the standalone non-voting window for hosts that certify decisions themselves. It stores `configuration_id`, a ring of `Learner_Cell(Value)` (`slot`, `value`; slot zero marks an empty cell), and `released_through`.

Proc	Contract
`learner_init(l, configuration_id: u64) -> Error`	`.Invalid_Configuration_Id` for zero.
`learner_learn_chosen(l, configuration_id: u64, slot: Slot, value: Value) -> (Learn_Result, Error)` (`learner_step`)	Record one certified value. Errors: `.Configuration_Mismatch`, `.Invalid_Slot`, `.Conflicting_Chosen_Value`, `.Window_Full` when `slot - released_through > MAX_ENTRIES`.
`learner_read_chosen(l, from_slot: Slot, output: []Chosen_Value(Value)) -> (int, Error)` (`learner_read`)	Copy the released suffix; `.Trimmed` once the ring has wrapped past it.
`learner_chosen_at(l, slot: Slot) -> (Value, bool)` (`learner_get`)	One released value still resident.
`learner_cell_index(slot: Slot, $MAX_ENTRIES: int) -> int`	`(slot - 1) mod MAX_ENTRIES`.

`Learn_Result` has three values: `.Buffered` (recorded, a gap below it holds release), `.Advanced` (the contiguous prefix moved past it), and `.Duplicate` (already released or buffered). `Chosen_Value(Value)` pairs `slot` and `value` in the output buffer.

12.6 Options, constants, and capacities

`Node_Options` is a plain struct whose zero value selects every default: `priority: u8` (breaks ballot ties, higher wins), `election_timeout_ticks`, `heartbeat_interval_ticks`, `resend_interval_ticks` (each `u32`, zero means the default), `gate_proposals_on_inherited_prefix: bool` (refuse proposals with `.Leader_Catching_Up` until the inherited prefix is delivered), `campaign_disabled: bool` (an acceptor that promises and votes but never campaigns), and `rotating_ownership: bool` (every member proposes in its own slots without phase one; campaigns are refused; stalls are repaired by bounded revocations).

Constant	Value	Role
`DEFAULT_MAX_MEMBERS`	7	Voter capacity of `Membership`, `Node`, `Effects`, `Stop_Sign`.
`MAX_SUPPORTED_MEMBERS`	65535	Ceiling on `MAX_MEMBERS`: member indexes and the ballot's node field are 16 bits.
`LINEAR_LOOKUP_LIMIT`	8	Memberships up to this size use a linear scan; larger ones binary-search the sorted members.
`DEFAULT_WINDOW_SLOTS`	256	Resident consensus cells per node; must be a power of two.
`DEFAULT_CHUNK_SLOTS`	64	Recovery chunk and batch bound; `1 <= CHUNK_SLOTS <= WINDOW_SLOTS`.
`DEFAULT_MAX_METADATA_BYTES`	256	Stop-sign metadata capacity.
`DEFAULT_MAX_ENTRIES`	256	`Learner` ring capacity.
`DEFAULT_ELECTION_TIMEOUT_TICKS`	10	Follower ticks without leader contact before campaigning; owner ticks without progress before a revocation.
`DEFAULT_HEARTBEAT_INTERVAL_TICKS`	3	Leader ticks between heartbeats.
`DEFAULT_RESEND_INTERVAL_TICKS`	10	Ticks between retransmission scans.
`SKIP_BURST`	8	Most no-op skips an idle owner sends per tick (also bounded by `CHUNK_SLOTS`).
`BALLOT_ROUND_BITS`	40	Width of the round field; `MAX_ROUND` is $2^{40} - 1$.
`WORD_BITS`	64	Bits per `Bit_Set` word.
`INVARIANT_CHECKS`	`ODIN_DEBUG`	`#config(PAXOS_INVARIANT_CHECKS, ODIN_DEBUG)`: internal assertions in debug builds; `-define:PAXOS_INVARIANT_CHECKS=true` enables them in release.

`Effects` is sized to the exact per-transition maxima, so the library never allocates and a full list is a bug rather than backpressure:

List	Capacity	Why the bound holds
`writes`	`2 * CHUNK_SLOTS + 1`	One transition drives at most one chunk of slots (`resolve_chunk`, `node_propose_batch`). Each slot costs one `Write_Vote`, followed by a `Write_Chosen` when the write quorum is one; `on_prepare` and `on_heartbeat` add at most one `Write_Promise`, and `promise_bounded` at most one `Write_Promise_At` per slot of the chunk.
`messages`	`MAX_MEMBERS * CHUNK_SLOTS + 2 * MAX_MEMBERS + 1`	One chunk per peer (accepts or commits for `CHUNK_SLOTS` slots to every peer, or `CHUNK_SLOTS` promises to

List	Capacity	Why the bound holds
		one candidate), plus one broadcast of prepares or heartbeats, plus one `Promise_Range_Message` or one `Learn_Message`.
`committed`	`WINDOW_SLOTS + 1`	`emit_contiguous` can release every resident cell, and `record_commit` can pass one further entry straight through when its cell is unavailable.
`requests`	`MAX_MEMBERS`	`on_learn` emits at most one `Serve_Range_Request` per transition; the bound allows one per peer.

12.7 State lifetime

A node begins in `.Follower` with `next_slot = leader_base = 1`. `node_campaign` (or an election timeout in `node_tick`) runs `start_campaign`: the ballot becomes `ballot_make(max(highest_observed_round, ballot_round(ballot), ballot_round(ledger_highest_ballot(ledger))) + 1, priority, id)`, the role becomes `.Preparing`, the election scratch (`election`, `promise_seen`, `recover_base`, the `recovered_*` and `lead_*` columns, `acknowledgements`, `acknowledged`) is cleared, and a `Prepare_Message` with `first = delivered_through + 1`, `last = slot_add(first, CHUNK_SLOTS - 1)`, and `scope = .Global` goes to every member. `maybe_resolve_chunk` waits until a read quorum has described the chunk completely; `resolve_chunk` re-drives every slot from the fence up; when no peer reports `more`, `become_leader` sets `.Leader`, `leader_hint = id`, `next_slot = leader_base = max(next_slot, slot_add(max(ledger_highest_used, fences), 1))`, and releases any contiguous prefix. A `Nack_Message` for the current ballot, with a greater promised ballot makes the node step down. A message handler can also call `observe_leader`, which demotes when the observed ballot differs from the local ballot. Its promise and role checks determine whether that call is reached.

Under rotating ownership `node_campaign` is refused; `tick_ownership` starts a `start_revocation` after `election_timeout_ticks` ticks without progress, sending a `.Bounded` prepare over the stalled chunk; `become_leader` then returns the node to `.Follower` once the chunk is driven, so there is no standing leader.

Entry	Keeps	Clears or rebuilds
`node_restore`	`promised`, `anchor`, every chosen cell, and every open vote above the base, which is the greater of `floor` and the anchor's `chosen_trim_slot`.	Open votes at or below the base; role, ballot, hint, timers, and the election scratch. `memory_floor` and `delivered_through` become the base; `next_slot` and `leader_base` become one above the highest used slot or the base; `own_next` is recomputed.
`node_continue_at`	Only the inherited `anchor`, which must not exceed `floor`; otherwise `.Trim_Regression`.	Everything else; the window resumes at `floor + 1`.
`node_begin_recovery`	`id`, `membership`, options, `promised`, and every cell above the anchor's `chosen_trim_slot`.	Cells at or below the anchor, the leader hint, timers, `peer_decided_through`, `resend_cursor`, and the election scratch; the role becomes `.Follower`. The anchor must pass the regression checks of `ledger_apply`.

Entry	Keeps	Clears or rebuilds
`node_restore_learner`	The decision-only `ledger` and its anchor.	Voting state; the floor is the anchor's `chosen_trim_slot`.

The replicated-log wrappers add `configuration_id`, reset the seal fields (`stop_sign`, `stop_slot`, `stop_pending`), and, for `restore`, `restore_learner`, and `begin_recovery`, rediscover a decided seal by scanning chosen cells for a stop sign that names a newer configuration.

12.8 Error reference

`Error` has forty-two values besides `.None`, grouped below as in `src/errors.odin`. `explain_error(err)` returns an operator-facing block with the problem and a hint. Class: **input** means the call was wrong and nothing changed; **backpressure** means retry later; **terminal** means this log or epoch cannot continue; **incident** means a safety assumption broke and the node must stop with its evidence preserved.

Value	Class	Meaning
Membership		
`.Empty_Membership`	input	No voter ids were given.
`.Too_Many_Members`	input	More ids than `MAX_MEMBERS`.
`.Invalid_Node_Id`	input	Id zero is the reserved sentinel.
`.Duplicate_Node_Id`	input	One id appears twice.
`.Invalid_Read_Quorum`	input	Override outside `1..=N`.
`.Invalid_Write_Quorum`	input	Override outside `1..=N`.
`.Non_Intersecting_Quorums`	input	`read + write <= N`; a phase-one quorum could miss a phase-two quorum.
Input and addressing		
`.Not_Member`	input	Sender, peer, or local id is outside the membership.
`.Wrong_Recipient`	input	`envelope.to` is not this node.
`.Invalid_Peer`	input	`node_reconnected` targeted the local node.
`.Invalid_Slot`	input	Slot zero; a prepare with `last < first`; a `Learn_Message` count outside `1..=CHUNK_SLOTS`; a floor or trim claim above `delivered_through`.
`.Read_Buffer_Too_Small`	input	Output cannot hold `decided_through - from_slot + 1` entries.
`.Unknown_Node`	input	Reserved for host routers; the core never returns it.
Role and capability		
`.Not_Voter`	input	A campaign or proposal on a learner.
`.Not_Learner`	input	`node_learn_chosen` on a voter.
`.Learner_Is_Voter`	input	Learner id lies inside the membership.
`.Learner_Message_Forbidden`	input	A learner received anything but a commit.
`.Configuration_Mismatch`	input	`Log_Envelope` id differs from the local one; the batch is empty.
Liveness and progress		
`.Not_Leader`	backpressure	Phase one has not completed for this ballot.

Value	Class	Meaning
`.Leader_Catching_Up`	backpressure	Inherited slots below `leader_base` are still undelivered (only with the gate option).
`.Window_Full`	backpressure	`next_slot - memory_floor` (or the next own slot) would exceed `WINDOW_SLOTS`, or a learner slot is beyond its ring.
`.Global_Slot_Exhausted`	terminal	The `u64` slot line is spent; it never wraps.
`.Empty_Batch`	input	A batch with no values.
`.Slot_Buffer_Too_Small`	input	Fewer output slots than values.
`.Ballot_Exhausted`	terminal	No round above `MAX_ROUND`.
`.Invalid_Promise`	input	A `Promise_Message` with `state = .Empty`, or a `Promise_Range_Message` with `last < first`, a wrong chunk limit, or `reported > CHUNK_SLOTS`.
`.Missing_Noop`	input	A chunk resolved without a no-op from `campaign` or `tick`.
`.Missing_Proposed_Value`	incident	A quorum acknowledged a slot whose cell the leader holds with no vote (a stale acknowledgement for a cell that moved on is ignored instead).
`.Campaign_Disabled`	backpressure	This voter never starts elections, or runs rotating ownership.
Durability and safety		
`.Promise_Regression`	incident	Replay moved a promise, or recorded a vote, below the durable promise.
`.Conflicting_Value`	incident	One ballot and slot carry two values.
`.Conflicting_Commit`	incident	One slot observed two chosen values.
`.Conflicting_Chosen_Value`	incident	A learner saw two chosen values for one slot.
`.Trim_Regression`	incident	A trim anchor moved backward or contradicts the adopted one.
Replicated log and window		
`.Invalid_Configuration_Id`	input	Configuration id zero.
`.Metadata_Too_Large`	input	Stop metadata above `MAX_METADATA_BYTES`.
`.Log_Sealed`	backpressure	A stop sign is pending or decided; finish the handover.
`.Batch_Too_Large`	input	More than `CHUNK_SLOTS` values.
`.Configuration_Id_Regression`	input	The next id is not greater than the current one.
`.Configuration_Id_Exhausted`	terminal	Reserved for hosts that allocate ids; the library never returns it.
`.Window_Overrun`	incident	A record, or a leader's own proposal, addresses a cell still holding an earlier open slot.
`.Trimmed`	backpressure	The slot was released at or below the floor; read it from the host journal.

12.9 Formula sheet

Quantity	Formula	Where it lives
Majority	$\lfloor N/2 \rfloor + 1$	`membership_init`: `total / 2 + 1`.
Crashes a majority tolerates	$N - (\lfloor N/2 \rfloor + 1)$	Five voters survive two.
Flexible quorum safety	$\lvert Q_1 \rvert + \lvert Q_2 \rvert > N$	`membership_init` refuses otherwise.
Ballot packing	`round << 24 | priority << 16 | node`	`ballot_make`; 40, 8, and 16 bits, so `<` on `Ballot` is B1.
Cell index	`(slot - 1) & (W - 1)`	`cell_of`; `W` is `WINDOW_SLOTS`, a power of two. `learner_cell_index` uses `(slot - 1) mod MAX_ENTRIES`.
Recovery index	`slot - recover_base`	`recovery_index` checks range before subtraction; scratch has `CHUNK_SLOTS` entries and does not use the ledger mask.
Owner of a slot	$(\text{slot} - 1) \bmod N$	`owner_of`, an index into the membership order; the owner's ballot is `ballot_make(0, 0, owner)`.
Stable-path messages per value	$3(N - 1)$	`send_accept` broadcasts $N - 1$ accepts; each `on_accept` answers once; `on_accepted` broadcasts $N - 1$ commits exactly once.
Acknowledgements a commit waits for	$\lvert Q_2 \rvert - 1$ replies	`send_accept` counts the leader's own vote first.
Window backpressure	$\text{next\_slot} - \text{memory\_floor} \le \text{WINDOW\_SLOTS}$	`node_propose`. A batch must fit in the free cells: `WINDOW_SLOTS` minus the occupied count `next_slot - 1 - memory_floor`.
Recovery chunk	$[\text{first}, \text{first} + \text{CHUNK\_SLOTS} - 1]$	`start_campaign`, `begin_next_chunk`; `slot_add` saturates at `max(Slot)`. A revocation caps `last` at `highest_seen`.
Effects capacities	$2C + 1$, $MC + 2M + 1$, $W + 1$, M	Writes, messages, committed, requests; C, M, W are the chunk, member, and window bounds.

12.10 Invariants for review

1. Ballots are unique and totally ordered: `Ballot` is one packed `u64` compared as an integer, round above priority above node; `start_campaign` stamps `node.id` and picks a round above every round the node has promised, voted, or observed; round zero is reserved for slot owners.
2. Every read quorum meets every write quorum: `membership_init` returns `.Non_Intersecting_Quorums` unless `read + write > N`.
3. No vote below the promise: `on_accept` answers a ballot below `promised` or below `promised_at[cell]` with a nack; `ledger_apply` refuses a `Write_Vote` below either with `.Promise_Regression`.
4. One value per ballot and slot: `on_accept` and `ledger_apply` return `.Conflicting_Value`; `send_accept` never replaces a live proposal at the same ballot with another value.
5. The greatest vote wins: `on_promise` keeps the highest-ballot vote per slot, and a reported decision dominates; `resolve_chunk` re-proposes it, or the no-op for a true hole, only after `maybe_resolve_chunk` has a read quorum of complete chunk descriptions. `recovery_ready` freezes the selected values before phase two, including across retries at the window boundary.
6. Chosen means a write quorum of distinct voters: `on_accepted` inserts the member index into `acknowledgements[cell]` and counts `acknowledged[cell]` against `membership_write_quorum`; a duplicate never counts twice.
7. A decided slot never changes: `record_commit` and `ledger_apply` return `.Conflicting_Commit`.
8. Release is a contiguous prefix: `emit_contiguous` advances `delivered_through` one slot at a time and stops at the first gap.
9. Persist before send: `effects_add_write` raises `writes_pending`; `effects_messages_slice` and `effects_reset` stop the process while it is raised under `.Enforced`.
10. Within a configuration, a decided slot is never assigned a second value, and live cells are retagged only for released history: `claim_live` reuses a cell only when its old slot is chosen and at or below `memory_floor`; `ledger_claim` also accepts slots at or below the anchor; `resolve_chunk` starts above `quorum_fences`.

12.11 Answers to selected exercises

12.11.1 Exercise 1.1

With voters `{1, 2, 3, 4}`, the sets `{1, 2}` and `{3, 4}` share no member. Suppose both were legal quorums. Node 1 campaigns with ballot `(1, 0, 1)` while a partition separates `{1, 2}` from `{3, 4}`; nodes 1 and 2 promise, node 1 proposes `x` for slot 1, and both vote, so `x` is chosen by `{1, 2}`. Meanwhile node 3 campaigns with `(1, 0, 3)`; nodes 3 and 4 promise, having voted for nothing, and node 3 proposes `y`; both vote and `y` is chosen by `{3, 4}`. Slot 1 now has two chosen values: agreement fails, because no member of the second quorum could carry the first quorum's vote into phase one. The library refuses the configuration before any of this can happen: `membership_init` with overrides `2, 2` returns `.Non_Intersecting_Quorums` since `2 + 2 <= 4`, and the default majority of four is three.

12.11.2 Exercise 2.1

`ballot_make` packs the round into bits 63 through 24, the priority into bits 23 through 16, and the node into bits 15 through 0, so integer comparison on `Ballot` orders round, then priority, then node. The ascending order is `(1, 5, 1) < (1, 5, 3) < (2, 0, 1) < (2, 0, 3)` and `(2, 0, 3)` wins. Round sits first because it is the freshness counter: every campaign takes a round above everything the node has seen, so a new attempt always outranks an old one, whatever its priority. Priority sits second so an operator can prefer one node within a round without ever letting a stale high-priority ballot outlive fresh ones; if priority came first, a promise to a crashed high-priority node would block every lower-priority candidate at every future round. The node id comes last purely to make ballots unique.

12.11.3 Exercise 4.1

The promise for `(5, 0, 2)` never became durable, so on restart `ledger_replay_fold` rebuilds `promised` without it, and nothing about `(5, 0, 2)` was ever transmitted: under `.Enforced`, `effects_messages_slice` stops the process while a write is pending, so the `Promise_Range_Message` for ballot `(5, 0, 2)` could not have left. No peer holds

evidence of the lost promise. `on_prepare` therefore compares `(4, 0, 3)` with the replayed promise, finds it not lower, records `Write_Promise{(4, 0, 3)}`, and answers with the chunk. The deciding rule is the first of the host contract: every write of a transition is durable before any message of that transition reaches a peer. Had the sync completed, replay would restore `(5, 0, 2)` and `on_prepare` would send `Nack_Message{rejected = (4, 0, 3), promised = (5, 0, 2), slot = 0}` instead.

12.11.4 Exercise 4.2

Claim: if a write quorum Q voted for v at ballot b in slot s, then every `Accept_Message` for s at any ballot $b' > b$ carries v. Induct on b' in ballot order, assuming the claim for every ballot strictly between b and b'. The leader at b' proposes for s only after `maybe_resolve_chunk` saw complete chunk descriptions from a read quorum R. By `membership_init`, R and Q share some acceptor a. Acceptor a's promise to b' was recorded after its vote at b: had the promise come first, `on_accept` would have nacked ballot $b < b'$, because `promised` never decreases (`ledger_apply` rejects `.Promise_Regression`). So when a answered `Prepare_Message` for b', its cell for s held a vote at some ballot c with $b \le c < b'$: at least b, since `on_accept` only replaces a vote with a higher ballot, and below b', since after promising b' it accepts nothing lower and the promise reflects the state at that moment. If $c = b$ the value is v; if $b < c < b'$ the induction hypothesis says the accept at c carried v. `on_promise` keeps the highest-ballot vote per slot across R; that highest ballot is at least c and below b', so by the same argument its value is v, and `resolve_chunk` re-proposes exactly that value. The no-op branch is unreachable because a reported a vote, and a cell reported with `state = .Chosen` names a chosen value, which is v by agreement.

12.11.5 Exercise 8.1

Write A_3 for the accept at `(3, 0, 1)` with value x and A_4 for the accept at `(4, 0, 2)` with value y. Each acceptor sees one of two orders. A_3 then A_4: `on_accept` finds `(3, 0, 1)` not below its promise, records `Write_Vote`, answers `Accepted_Message` to node 1; then `(4, 0, 2)` is not below `(3, 0, 1)`, so it overwrites the vote, records another `Write_Vote`, and answers `Accepted_Message` to node 2. A_4 then A_3: it votes y and raises the cell's `promised_at` to `(4, 0, 2)`; then `(3, 0, 1)` is below that promise, so it sends `Nack_Message{rejected = (3, 0, 1), promised = (4, 0, 2)}` with the slot, and node 1, on `on_nack`, drops to `.Follower`. An acceptor that already promised `(4, 0, 2)` during node 2's phase one nacks A_3 in either order. Every acceptor that receives A_4 at all votes y, so y is chosen in every arrival order once two acceptors receive it. x could be chosen only if two acceptors saw A_3 before promising `(4, 0, 2)`; but node 2 needed promises from two acceptors, one of which would then have reported x at ballot `(3, 0, 1)`, and `resolve_chunk` would have made A_4 carry x, contradicting the premise. So only y can be chosen, and if x ever reached a quorum both accepts would carry the same value.

12.11.6 Exercise 11.1

Say slots 1 through 9 are delivered. The next candidate sends `Prepare_Message{first = 10, last = 10 + CHUNK_SLOTS - 1, scope = .Global}`. The acceptor holding the slot-10 vote answers `Promise_Message{slot = 10, state = .Voted}` and a `Promise_Range_Message` with `reported = 1`; the one holding slot 12 answers likewise for 12; the others report `reported = 0`. Once a read quorum has described the chunk, `resolve_chunk` computes the fence as 9, starts at slot 10, and bounds the drive at `known_high = 12`. Slot 10 has a recovered vote (`recovered_state = .Voted`), so `send_accept` re-proposes its value under the new ballot; slot 11 has no vote, so `send_accept` carries the campaign's no-op; slot 12 re-proposes the recovered value. `become_leader` then sets `next_slot = leader_base = 13`. If the read quorum happens to miss the single acceptor that voted in 10 or 12, that slot looks like a hole and gets the no-op too, which is safe because a lone vote is not a choice; if the old leader's own durable vote is in the quorum, it is reported like any other.

12.11.7 Exercise 14.1

Elections with two of five down need $Q_1 \le 3$; safety needs $Q_1 + Q_2 > 5$, so $Q_2 \ge 3$. The cheapest commit compatible with both is $Q_1 = Q_2 = 3$, the majority pair, which `membership_init(&m, ids)` selects by default and `3 + 3 = 6 > 5` satisfies. A commit then waits for two `Accepted_Messages` beyond the leader's own vote, and message count is unchanged at $3(N-1) = 12$ per value, since `broadcast_peers` always writes to every peer;

quorum size changes latency, not traffic. The genuinely cheaper setting $Q_2 = 2$ forces $Q_1 = 4$: a commit waits for one reply, but an election needs four live voters, so a single crash is the most the cluster can lose and still elect.

12.12 Glossary

Term	Meaning
Ballot	One `u64` packing `(round, priority, node)`, the unique, totally ordered name of one proposal attempt.
Ledger	`Ledger(Value, WINDOW)`: the acceptor's durable state in Lamport's variables, laid out as struct-of-arrays and rebuilt by replaying `Write` records.
Cell	One index of the window, `cell_of(slot, WINDOW)`, tagged by `slot[c]` with the slot it currently holds and by `state[c]` with `.Empty`, `.Voted`, or `.Chosen`.
Chosen	A write quorum has durably voted for one value in one slot; true before anyone knows it. A cell whose `state` is `.Chosen` records a known decision (`Write_Chosen`).
Applied	The host state machine has executed a released `Committed` entry.
`decided_through`	The greatest slot released in contiguous order; the field is `delivered_through`.
Memory floor	The greatest slot the host has durably consumed; cells at or below it may be retagged.
Trim anchor	`Trim_Anchor{trim_id, chosen_trim_slot}`: every slot at or below it is chosen and folded into the host image the host binds to `trim_id`.
Fence	`quorum_fences`: the greatest trim anchor and chosen prefix a read quorum reported (`Fences{trim, chosen, chosen_peer}`); recovery starts above it.
Chunk	`CHUNK_SLOTS` consecutive slots answered by one `Prepare_Message`, or proposed by one batch.
Bounded prepare	A `Prepare_Message` with `scope = .Bounded`: the acceptor promises only the decrees in `[first, last]`, each as a `Write_Promise_At`.
Owner	Under rotating ownership, the member `owner_of(node, slot)` that may propose in `slot` at round zero without phase one.
Suggestion	An owner's proposal in its own slot at `ownership_ballot(owner)`.
Skip	A no-op an idle owner proposes in its own slot below `highest_seen` so the log never waits on it; at most `SKIP_BURST` per tick.
Revocation	A bounded phase one over a stalled chunk at a round above zero, fencing the owner out of those slots only and re-proposing any vote found.
Resubmission	An owner proposing again, in its next own slot, a suggestion that a revocation decided to another value.
Stop sign	A decided `Stop_Sign` entry that seals its configuration and names the next one.
Configuration id	The host-allocated, strictly increasing `u64` naming one voter set.
Term base	`leader_base`: the first slot the current leadership may fill; everything below was inherited.
Learner	A non-voting participant that only receives commits or certified decisions.
No-op	The host's harmless value, remembered from `campaign` or `tick`, that fills a recovered hole or a skipped slot.
Quorum	Any subset of voters of the required size.

Term	Meaning
Read and write quorum	`read_quorum_size` for phase one, `write_quorum_size` for phase two; their sum exceeds N.
Envelope	`Envelope{from, to, message}`; `Log_Envelope` adds `configuration_id`.
Effect batch	One `Effects` value: the writes, messages, committed entries, and requests of one transition.

12.13 Sources

1. Leslie Lamport, "The Part-Time Parliament", *ACM Transactions on Computer Systems* 16(2), 1998. The parliament metaphor, the ledger, B1 through B3, and the multi-decree refinements this book follows.
2. Leslie Lamport, "Paxos Made Simple", *ACM SIGACT News* 32(4), 2001. The same protocol derived from the safety requirement in plain prose.
3. Leslie Lamport, Dahlia Malkhi, and Lidong Zhou, "Reconfiguring a State Machine", *ACM SIGACT News* 41(1), 2010. Stop signs and configuration handover on one slot line.
4. Yanhua Mao, Flavio P. Junqueira, and Keith Marzullo, "Mencius: Building Efficient Replicated State Machines for WANs", OSDI 2008. Rotating slot ownership, skips, and revocation, which `src/ownership.odin` follows.
5. Heidi Howard, Dahlia Malkhi, and Alexander Spiegelman, "Flexible Paxos: Quorum Intersection Revisited", OPODIS 2016. Why only phase-one and phase-two quorums need to intersect.
6. Robbert van Renesse and Deniz Altinbuken, "Paxos Made Moderately Complex", *ACM Computing Surveys* 47(3), 2015. Slots, ballots, and the roles of a complete implementation.
7. The Odin language overview, odin-lang.org/docs/overview. Parametric structs, unions, `bit_set`, `Maybe`, and `or_return`, the language features the library leans on.

12.14 Closing note

Paxos is one preservation rule carried through time. Quorum intersection guarantees a witness; stable storage lets the witness remember; phase one asks the witnesses; phase two records; slots put decisions in order. Everything in `src/` is that rule made concrete, and everything the host does with sync, identity, and deterministic application is that rule made physical.

Review & Discussion

Summary Reflection: Synthesize the Paxos protocol in terms of its five core invariants:

1. Lexicographical ballot uniqueness (B1).
2. Overlapping phase-one and phase-two quorums (B2).
3. Preservation of the highest-ballot reported vote during candidate election (B3).
4. Write-ahead durability ordering on promises and votes.
5. Contiguous prefix delivery and safe memory window reuse.

PART VIII

Conformance

One table from the paper to the code to the oracles: every safety-relevant rule of the basic protocol is traceable to a proc, and every oracle names the invariant it watches.

13 Lamport Conformance Appendix

This appendix maps the basic protocol of *The Part-Time Parliament* (section 2.3) and its multi-decree refinements (sections 3.1 through 3.3) onto `src/election.odin`, `src/consensus.odin`, `src/ownership.odin`, `src/ledger.odin`, and `src/replicated_log.odin`, and then onto the runtime oracles in `sim/simulation.odin`, the seeded reconfiguration scenarios in `tests/test_reconfiguration_sim.odin`, and the ownership scenarios in `tests/test_ownership.odin`. Line numbers drift; proc names are the stable anchors. Read the whole chapter with one caveat in mind: what follows is executable evidence that the implementation follows the paper on every schedule the harnesses have run. It is not a proof. The argument that B1, B2, and B3 imply agreement, and the lemmas that tie each proc to one of the three, live in the safety-argument chapter (`docs/book/03_proofs.typ`); this appendix only says where each lemma's premise is enforced and which oracle would notice if it were not.

13.1 The basic protocol, step by step

The paper's priest keeps `lastTried`, `nextBal`, `prevVote`, and the ledger. The library's acceptor keeps a `Ledger`: `promised` for every decree, a per-decree `promised_at`, one slot-tagged cell per resident decree, and the volatile `node.ballot` for its own attempts.

Paper	Rule	Implementing proc
Step 1	A priest chooses a ballot greater than `lastTried`, owned by itself, and sends `NextBallot(b)`.	`start_campaign`: the round is one above the greatest of `highest_observed_round`, `ballot_round(ballot)`, and the round of `ledger_highest_ballot`, packed by `ballot_make` with the node's `priority` and `id`, so ownership is built into the ballot; the `Prepare_Message` with `scope = .Global` goes to every member.
Step 2	On `NextBallot(b)` with `b` above `nextBal`, set `nextBal` and answer `LastVote(b, v)` with the highest vote below `b`.	`on_prepare`: a ballot below `promised` gets a `Nack_Message`; otherwise `Write_Promise` is recorded (or, for `scope = .Bounded`, `promise_bounded` records one `Write_Promise_At` per decree in the range), one `Promise_Message` per used cell in the chunk is sent with its `vote` and `state`, and a `Promise_Range_Message` closes the answer.
Step 3	With `LastVote` from every priest in a majority, begin the ballot with the decree of the highest vote, or any decree if none (B3).	`on_promise` keeps the highest-ballot vote per slot, lets a reported decision dominate, and refuses two values at one ballot; `on_promise_range` records each peer's chunk description; `maybe_resolve_chunk` waits for `membership_read_quorum` complete descriptions; `resolve_chunk` re-proposes each recovered value above the quorum fences.
Step 4	On `BeginBallot(b, d)` with `b` equal to `nextBal`, cast the vote and write it in the ledger.	`send_accept` records the proposer's own `Write_Vote` and broadcasts `Accept_Message`; `on_accept` nacks a ballot below `promised` or below the cell's `promised_at`, otherwise records `Write_Vote` and answers `Accepted_Message` only after that write is in the batch.
Step 5	With `Voted` from every priest in the quorum, the decree passes and the president writes it in the ledger.	`on_accepted`: acknowledgements are a `Bit_Set` keyed by member index, so duplicates never count twice; at `membership_write_quorum`, `record_commit` writes `Write_Chosen` and `Commit_Message` is broadcast once.

Paper	Rule	Implementing proc
Step 6	On `Success(d)`, every priest writes the decree in its ledger.	`on_commit` and `record_commit`: `.Conflicting_Commit` if the slot holds a different decision; `emit_contiguous` then releases the decided prefix in order.

Several departures from the paper's message rules are deliberate. A repeated Prepare at the current promise is answered idempotently. An Accept may carry a ballot above the effective promise; accepting it raises the per-slot promise. A repeated Accept for the same vote needs no new write. `on_prepare` reports a decided cell in the same answer as the votes, with `state = .Chosen`, and `on_promise` lets it dominate every vote: the paper's president learns passed decrees separately. And round zero of every decree is reserved for the decree's owner (`ownership_ballot`), which `on_accept` enforces by refusing a round-zero accept from anyone else; under rotating ownership an owner therefore runs step 4 in its own slots with no step 1 at all, because no lower ballot exists in that decree, and the Synod proof applies unchanged.

13.2 Multi-decree refinements

Paper	Refinement	Implementing code
Section 3.1	One `NextBallot(b, n)` covers every decree numbered above `n`; the reply carries all of that priest's votes for those decrees.	`Prepare_Message.first` is the paper's `n + 1` and `scope = .Global` promises every decree from it on; `on_prepare` reports only `[first, last]`, one chunk of `CHUNK_SLOTS`, and sets `Promise_Range_Message.more` when used cells exist above `last`; `begin_next_chunk` issues the next `Prepare_Message` under the same ballot. Leadership, once won, covers every later slot without a new phase one.
Section 3.2	Gaps below a passed decree are filled with the harmless "olive-day" decree so the ledger stays contiguous.	`resolve_chunk` proposes the no-op remembered from `campaign` or `tick` for every slot in the drive range that has no recovered vote; slots at or below the quorum's trim anchor or chosen prefix are released history and are not treated as holes. Under ownership `skip_idle_slots` proposes the same no-op in an idle owner's slots before anyone has to recover them.
Section 3.3	A decree can change the membership; the change takes effect a fixed number of decrees later, so the parliament that passes it is the one that decides the intervening decrees.	`src/replicated_log.odin` uses a stop sign instead of a delay: a decided `Stop_Sign` in slot s seals its configuration, nothing above s is released to the application in it, proposals answer `.Log_Sealed`, and `replicated_log_init_from_stop` starts the next configuration at $s + 1$ on the same slot line with an inherited trim anchor. Under rotating ownership, decisions made above the seal before it was learned are abandoned; the new configuration decides those positions afresh.
Section 2.2 (B1)	Each ballot has a unique number.	`Ballot` is one packed `u64`, `round << 24 | priority << 16 | node`, so the total order is integer comparison and the node field makes two proposers'

Paper	Refinement	Implementing code
		ballots distinct; per decree, round zero belongs to the owner alone.
Section 2.2 (B2)	Any two quorums share a priest.	`membership_init` rejects `read + write <= N` with `.Non_Intersecting_Quorums`; majority by default, flexible pairs allowed.
Section 2.2 (B3)	A ballot's decree equals the decree of the highest earlier vote in its quorum, if any.	`on_promise` and `resolve_chunk`, checked by `test_lamport_b3_max_vote_rule` in `tests/test_protocol.odin` and by `ownership_revocation_keeps_a_seen_vote` in `tests/test_ownership.odin`.

13.3 Durable state: the ledger

The paper's ledger variables map onto the columns of `Ledger(Value, WINDOW)`. The host never copies the struct; it persists the five `Write` records and rebuilds the struct by replay. Cells are struct-of-arrays, and `slot[c]` tags which slot a cell holds so a reused cell is never mistaken for an older one.

Paper	Column	Persisted by
`nextBal`	`promised`, and per decree `promised_at[c]`; the effective promise is `ledger_promise_for`	`Write_Promise` from `on_prepare` (`.Global`) and `on_heartbeat`; `Write_Promise_At` from `promise_bounded`. A vote also raises `promised_at[c]` to its ballot.
`prevVote`	`vote_ballot[c]` and `value[c]` while `state[c]` is `.Voted`, read by `ledger_vote_at`	`Write_Vote` from `send_accept` and `on_accept`, before the `Accepted_Message` may be sent.
outcome	`value[c]` while `state[c]` is `.Chosen`, read by `ledger_chosen_at`	`Write_Chosen` from `record_commit`, before the commit broadcast or application delivery.
`lastTried`	`node.ballot` (volatile)	Not persisted: a restarted node campaigns above `ledger_highest_ballot`, which is safe because every accept it ever sent at a campaign ballot implies its promise was durable first, and an owner's round-zero suggestion waits for the barrier.
(none)	`anchor`	`Write_Trim` from `node_install_chosen_trim`; `node_begin_recovery` applies one through `ledger_apply` and emits nothing, so the host persists the image and the anchor itself. The paper predates bounded windows; the anchor lets an acceptor vouch for a released prefix it no longer stores.
(derived)	`used`, `chosen`	Bitmaps over cells, maintained by `ledger_record_vote` and `ledger_record_chosen`; never persisted.

`ledger_apply` is the strict single-configuration replay and the rule set a reviewer should check against the paper:

- `Write_Promise` below `promised` is `.Promise_Regression`; otherwise it replaces the promise.
- `Write_Promise_At` at slot zero is `.Invalid_Slot`; a cell still holding an earlier open slot is `.Window_Overrun`; a ballot below the cell's `promised_at` is `.Promise_Regression`; otherwise it replaces the per-decree promise.

- `Write_Vote` at slot zero is `.Invalid_Slot`; below `promised` it is `.Promise_Regression`; a cell still holding an earlier open slot is `.Window_Overrun`; below the cell's `promised_at` it is `.Promise_Regression`; the same ballot with a different value is `.Conflicting_Value`; a value that disagrees with a stored decision is `.Conflicting_Commit`; otherwise `promised_at[c]` follows the ballot and, unless the cell is already `.Chosen`, the vote is stored.
- `Write_Chosen` at slot zero is `.Invalid_Slot`; one whose cell a later slot already owns is silently skipped, because a decision is derived state that the anchor or a later decision already covers; one that disagrees with a stored decision is `.Conflicting_Commit`; otherwise the decision is stored.
- In phase one, `on_promise` keeps per decree the greatest vote reported and lets a reported decision dominate; a `.Voted` report that arrives after a decision is an older, losing vote from an acceptor outside the deciding quorum and is ignored, and only a second decision with a different value is `.Conflicting_Commit`.
- `Write_Trim` with a lower `trim_id` or lower `chosen_trim_slot`, or the same non-zero `trim_id` with a different slot, is `.Trim_Regression`.

A decision that disagrees with a stale local **vote** is legal and accepted; the choosing quorum may not have included this node. `ledger_replay_fold` relaxes the rules for lifetime journals that span window reuse and configuration changes: a promise folds to its maximum, a per-decree promise or a vote whose cell is already held by a later slot is skipped rather than reported as an overrun, a vote is stored without the global-promise check, and decisions and anchors go through `ledger_apply` unchanged. `ledger_claim` decides cell reuse during replay: a cell may be retagged when its previous slot is chosen or lies at or below the certified trim.

13.4 The oracles that watch the same rules

The simulator in `sim/simulation.odin` runs up to five nodes (`MAX_SIM_NODES`) with a 256-slot window and 64-slot chunks (`SIM_WINDOW`, `SIM_CHUNK`). One seed fixes every choice; a failing run prints the seed so it can be replayed step for step with `./bin/paxos-sim --seed=N --steps=N --nodes=N [--ownership] [--verbose]`, and `make sim` runs one seeded pass. The default fault mix (`default_faults`) drops 60 per thousand envelopes, duplicates 40, crashes 20 per transition, and toggles a link 25 per thousand steps. The harness carries values the way a host would: `packet_of` copies a message's value at enqueue and `packet_envelope` repoints the envelope at that copy before `step`, and every journaled `Sim_Record` copies the value of a `Write_Vote` or `Write_Chosen`.

The simulator has two modes. Without `--ownership`, node 1 campaigns at bootstrap and again during quiescence, and the liveness probe is proposed by whichever node reports `.Leader`. With `--ownership`, every node is initialised with `Node_Options{rotating_ownership = true}`, nobody campaigns, a proposal on any live node succeeds in that node's own slot, and the probe is proposed by the first node the loop reaches. The oracles are the same in both modes.

Crashes strike inside the host commit sequence at one of three `Crash_Points`: `.Before_Writes` (nothing survives), `.Partial_Writes` (a prefix of the writes is durable, no message left), and `.Partial_Messages` (every write durable, a prefix of the messages sent). Before the crash roll, the accepts that `pre_durable_next` yields are enqueued, so a campaign accept can reach a peer whose sender then loses its batch. Restart replays the journal through `ledger_replay_fold` and `restore` with the consumed prefix as floor and the same `Node_Options`. `advance_memory_floor` is called after only half of the transitions, so full-window paths are exercised.

Oracle	What it checks	Proc	Invariant witnessed
Agreement	The first decision per slot fixes the golden log; any later decision for that slot must equal it. A decision is recorded at the vote level: a write quo-	`record_decision`, `persist_sim_write`	One value per slot; chosen means voted.

Oracle	What it checks	Proc	Invariant witnessed
	rum of `Write_Vote` records at one ballot counts as a decision even if the proposer crashes before announcing it, and every `Write_Chosen` and every released `Committed` entry is checked against the golden log.		
Validity	Every decided value is the no-op or a value some node proposed in this run.	`record_decision`	Values come from proposals.
Promise monotonicity	A `Write_Promise` never carries a ballot below the node's last durable promise.	`persist_sim_write`	`nextBal` is monotone.
Vote below promise	A `Write_Vote` never carries a ballot below the node's global promise, and one ballot never records two values in a slot across the cluster.	`persist_sim_write`	Steps 2 and 4; one value per ballot.
Contiguity	Each released `Committed` entry is exactly the consumed prefix plus one.	`process_effects`	Contiguous delivery.
Liveness probe	After healing links, restarting every node, and draining the network, the cluster must decide one fresh proposal, so a run without progress cannot pass vacuously.	`run_quiescence`, `sim_run`	Progress under quiescence.
Convergence	Every node applied every slot of the golden log with the golden value.	`verify_convergence`	All ledgers agree at the end.

The seeded reconfiguration scenarios in `tests/test_reconfiguration_sim.odin` run a three-voter `Replicated_Log_Node` (capacity `SEAL_MEMBERS` of 4) with an eight-slot window, two-slot chunks, and 32 metadata bytes (`SEAL_WINDOW`, `SEAL_CHUNK`, `SEAL_METADATA`), shuffling delivery order per seed, stamping every envelope

through `log_envelope`, and journaling every write with `journal_append` before any message leaves. Each scenario runs sixteen seeds (`SEAL_SEEDS`) and applies the same oracles: `seal_expect_agreement` (every member decided the same stop sign in the same slot, with identical sealed prefixes), `seal_expect_nothing_after` (no slot past the seal reads as decided and proposals return `.Log_Sealed`), `seal_expect_replay_keeps_seal` (journal replay through `journal_replay` and `restore` rediscovers the seal), and `seal_expect_next_epoch_decides` (the configuration the stop sign names elects a leader and decides new commands at `stop_slot + 1` onward).

- `reconfiguration_sim_seal_survives_drop_duplicate_and_reorder`: members `{1, 2, 3}` keep the same voter set while the configuration id moves from 1 to 2. A command and the stop sign race for adjacent slots; the accept carrying the stop is dropped once toward node 2 and duplicated once toward node 3, and the test asserts both faults fired.
- `reconfiguration_sim_membership_handover_reaches_new_configuration`: `{1, 2, 3}` hands over to `{2, 3, 4}`, id 1 to 2. The removed voter's `log_init_from_stop` returns `.Not_Member`; node 2 leads the next configuration.
- `reconfiguration_sim_one_for_one_voter_replacement`: `{1, 2, 3}` becomes `{1, 2, 4}`, id 7 to 8. Node 3 leads the old configuration and is refused by the new one; node 1 leads after handover.
- `reconfiguration_sim_ownership_abandons_decisions_above_the_seal`: the same three members run with `rotating_ownership = true`. Owner 2 seals in slot 2 while owners 3 and 1, not yet aware of it, get slots 3 and 4 decided. The test requires that some ledger holds slot 3 as chosen, that no node releases or reads anything above the seal (`decided_through` stops at the stop slot, `read_decided` returns two entries), and that the next configuration decides slot 3 afresh. The host commit sequence in the harness carries the oracle for every scenario: a node sealed by a decided stop sign releases nothing above it.

The ownership scenarios in `tests/test_ownership.odin` run three `Node(u64, 3, 16, 4)` values with `rotating_ownership = true` and a queue that can silence one member as if it had crashed:

- `ownership_three_owners_propose_concurrently`: `owner_of` deals slots 1, 2, 3, then 4 to members 1, 2, 3, 1; four concurrent proposals decide in their own slots with no campaign, and `campaign` answers `.Campaign_Disabled`.
- `ownership_idle_owners_skip`: only member 1 proposes, in slots 1 and 4; after three ticks members 2 and 3 have skipped slots 2 and 3 with the no-op and every member has `decided_through` of 4.
- `ownership_revokes_a_crashed_owner`: member 3 is silent; the prefix stalls at 2 until a revocation fills slot 3 with the no-op, after which both survivors report `.Follower`, because a revoker holds no standing leadership.
- `ownership_revocation_keeps_a_seen_vote`: member 3's suggestion for slot 3 reached only member 2 before member 3 fell silent; the revoker finds that vote and re-proposes it (B3), so slot 3 decides 33, not the no-op.
- `ownership_revoked_suggestion_is_resubmitted`: member 3's suggestion reached nobody and is revoked to the no-op; when member 3 returns it learns the revocation and proposes the value again in a later own slot.

In all, `tests/*.odin` holds 79 `@(test)` procedures; the ones above are the schedule-driven subset. `review_hundred_twenty_eight_voters` and `review_thousand_voters_reach_quorum` in `tests/test_review.odin` exercise the sorted membership array and the word-array bit set at sizes the default capacities never reach.

13.5 What this evidence does and does not say

The mapping from paper step to proc is one-to-one with the first table, and each oracle names a proc and an invariant, so a change to any handler should update this appendix, the oracle, and the unit test that pins the rule together. The evidence is bounded by what was run: finitely many seeds, five nodes, one fault mix, two modes. A schedule the generator never produced is not covered, and no claim here rests on exhaustive state exploration. The lemmas in the safety-argument chapter remain the argument, the invariants in Part VII are its checklist, and the oracles are its instrumentation.

PART IX

Python integration

A Python SDK: a small interface over a precise durability contract. Follow one command across the language boundary, then examine what a timeout, a crash and a released buffer mean.

14 Paxodin: A Python Host for the Odin Core

Learning Objectives
After completing this chapter, you will be able to:
- Integrate Python applications with the Odin consensus core across a stable, versioned C ABI boundary.
- Implement durable file-backed journals and histories adhering to the strict persist-before-send contract.
- Track command lifecycles across the agreement, release, and application boundaries.
- Build asynchronous consensus workflows using `AsyncSession` and standard Python `asyncio`.
- Manage buffer memory lifetimes and wire codecs safely without introducing memory safety hazards.

Status. The package exists at `python/paxodin/` and is described by POD 0011. Its C ABI, typed `Node`, durable `Session`, asyncio `AsyncSession`, typed message classes, reference journal and history, wire codec and in-process clusters are implemented and covered by `make check-python`. A wheel builds from a source distribution outside the checkout and passes a three-node smoke test on CPython 3.12, 3.13 and 3.14 with no Odin compiler present. Reconfiguration, rotating ownership, learners and leases are refused by capability bit rather than half-supported. Tagged releases publish the tested wheels and source distribution to PyPI as `paxodin`.

14.1 Start with one command

Imagine three processes maintaining the same sequence of commands. A Python application asks one participant to append `b"set counter 41"`. The bytes become a value in the Odin log. The Python layer drives journal writes and message exchange until it can report the command's position in a durable, contiguous prefix. The application later interprets those bytes and updates its counter.

There are three distinct events here: agreement, release and application. An `append` receipt reports the first two at this participant. It does not say that every participant has received the command, or that the counter has changed. Keeping that distinction visible makes the API easier to reason about after a crash.

```
from paxodin import FileHistory, FileJournal, Session

# Supply an authenticated transport connecting this member to its peers.
with Session(
    node_id=1,
    members=[1, 2, 3],
    configuration_id=1,
    journal=FileJournal("state/node-1"),
    history=FileHistory("state/node-1"),
    transport=transport,
) as session:
    receipt = session.append(b"set counter 41", timeout=5.0)
    print(receipt.slot)
```

A Session represents one participant. The other two must also run and serve traffic. `poll(timeout=...)` keeps a participant progressing between appends. The initial design has no background worker. A follower reports `NotLeader` with a hint when available, leaving request routing explicit.

14.2 A small Python surface

The SDK uses Python 3.12+ features directly. Results are frozen dataclasses with slots; options are keyword-only; alternatives use union annotations; storage and transport adapters satisfy typed protocols. Public objects own their data. A Python `bytes` returned today remains valid after tomorrow's transitions.

The rule that shapes the surface is that Odin is the engine and Python is the product. Messages are nine typed classes an adapter matches on, never a tag and a field to consult. Timers are seconds; the engine's ticks are converted away. Errors render exactly as the core's do - a titled banner, the cause with its values, a `Hint:` - and a test enumerates every exception class as the Odin suite enumerates its `Error` enum. Reading the log is iteration.

`Session` is inspired by Requests in its context management, discoverable verbs, explicit timeouts and useful exceptions. `AsyncSession` is the same participant driven by asyncio: no polling loop, one lock owning every transition, journal syncs off the event loop, and a cancellation that stops waiting without pretending to un-propose. The subject is a replicated log, so completion rules come from the log. A third API, `Node`, exposes individual transitions and pending batches for hosts that need control over scheduling or storage. Both APIs invoke the same Odin core.

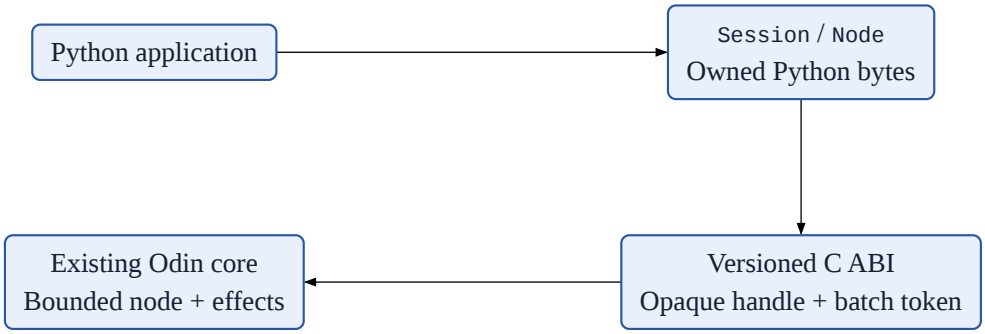


Figure 18: The SDK layers. Python owns returned data; the bridge hides native representation.

The initial wheel contains a fixed capacity profile. Python cannot instantiate an arbitrary Odin generic at runtime. The stock profile supports seven members, a 256-slot window, a 64-slot recovery chunk and commands up to 1,024 bytes. These limits are queryable and checked before mutation. Larger profiles need compatible artifacts on every participant.

14.3 Why the bridge retains a batch

Suppose Odin emits a vote and an outgoing acknowledgement. The wrapper starts constructing a Python list, but Python raises `MemoryError`. If that list were the only record of the pending writes, the next call could accidentally acknowledge a vote that never reached disk.

The bridge therefore retains the native effects batch behind a generation token. Python can ask for buffer sizes and retry copies without rerunning the transition. No new transition is permitted until that batch has been dealt with.

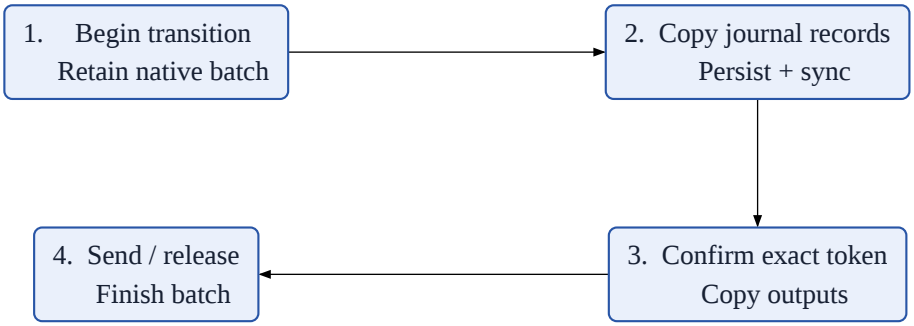


Figure 19: Write before send across two languages. A failed allocation leaves the batch pending.

The sequence has four steps. First, perform one transition and retain its effects. Second, copy the journal records, append them in order and sync. Third, confirm that exact batch token and copy outputs into owned Python storage. Finally, finish the batch and send or release the copied outputs. A protocol error may accompany writes, so checking the error never substitutes for processing the batch.

This design pays for copies at the language boundary. Those copies give ordinary Python lifetimes to values whose native storage is mutable. A later optimization must preserve that ownership contract. An exported pointer into the ledger would make a short API much harder to use correctly.

14.4 A timeout leaves a question open

A caller waits five seconds. The peers may have chosen the command while the reply was delayed. Raising `CommitTimeout` cannot undo that choice. The exception therefore reports whether admission occurred and, when known, the configuration and slot.

```
CommitTimeout: slot 28 was admitted, but its decision was not observed within 5s.
Hint: keep polling and inspect local history. This timeout did not cancel Paxos.
Use a command id and application deduplication before retrying into another slot.
```

The first sentence says what is known. The hint says how to make progress without inventing a guarantee. The SDK does not silently submit another copy after an admitted timeout. A receipt also does not create exactly-once application: a host still needs a durable command-id policy when its clients retry.

14.5 A memory floor transfers responsibility

A finite native window eventually fills. Freeing a cell is safe only after the host has durably taken responsibility for the released entry. In the implementation, Session writes released entries to retained history before moving the native memory floor. The application's durable cursor is separate: it records what the application has applied, not what Session has merely stored.

Position	What it establishes
Released prefix	This participant knows the decisions in order.
Native memory floor	The host durably retains the entries needed outside the window.
Application cursor	The application has durably consumed this prefix.

On restart, journal replay reconstructs the core. Retained history supplies unapplied commands and catch-up replies. V1 does not delete this history; bounded RAM is not bounded disk usage. Disk quotas stop admission with a useful error. Snapshot installation and history trimming require a later, explicit contract. A complete record with a bad checksum is corruption, not permission to skip it.

14.6 Packaging without a second algorithm

The project lives in `python/paxodin/`. uv manages its environment and locked development dependencies. hatchling is the build backend, and a build hook runs `odin build -build-mode:shared` directly; ctypes loads the result. There is no C or C++ in the project, so there is no C build system in it either. Wheels include the library, so ordinary wheel users need no Odin compiler. Source distributions include the exact core source revision and must build independently of the surrounding checkout.

Two libraries are built from one source. The shipped one compiles the core with `Host_Managed`, because a host inside a Python interpreter cannot accept a gate that calls `os.exit`; the bridge enforces the same order itself and returns a status. A second library compiles `Enforced`, and the whole test suite runs again against it, so an ordering mistake in the bridge stops a test run instead of reaching a release.

The development tools are Ruff, strict mypy, pytest and Hypothesis. The first release matrix covers CPython 3.12-3.14 on Linux x86-64, Windows x86-64 and macOS Apple Silicon. Each tagged release checks the installed artifacts before publishing. The release has to verify resource loading, native dependencies, portable CPU instructions, ABI versions and the absence of source-tree path assumptions. POD 0011 links the upstream tooling documentation and specifies the build gates.

14.7 Measure the wrapper's actual cost

The native benchmark measures neither Python object creation nor a Python journal, so the same workload was run through every path with the membership, payload, capacities and completion rule held equal. One member, one value per transition, the batch fully discharged and the memory floor advanced each time.

Path	ns per value	vs native	What it adds
native Odin	826	1.00	the transition alone
C ABI	7,008	8.52	`ctypes` crossings and copies
Python `Node`	23,787	28.78	owned Python objects
`Session`, memory	29,739	35.98	framing, journal, ordering

`Session`, `fsync`	48,091	58.29	a durable barrier per batch

Measured on an AMD Ryzen 7 5800H with CPython 3.13.5, nine samples per row, 64-byte payloads; raw data in `bench/results/paxodin-paths-20260917.json`.

Read the table as a distribution, not a ranking. The rows are not interchangeable, and comparing a Python `fsync` against a native in-memory transition would say nothing at all. What it shows is where the cost goes. The boundary itself is 883 nanoseconds per crossing at seven crossings per value, so the C ABI accounts for 6.2 of its 7.0 microseconds. Building owned Python objects costs a further 17 microseconds, and that is the ownership contract being paid for: a `bytes` handed to an application today stays correct after any number of later transitions.

Payload size moves the native and ABI rows by under five percent from eight bytes to 1,024, because values are stored inline at a fixed size and a larger one costs the engine no allocation - the same property that makes the node's footprint knowable in advance. The Python rows move by up to twelve percent at 1,024 bytes, which is the copy into an owned `bytes`, paid once per released entry.

This is what a measurement is for. Batched FFI would address 6.2 microseconds; a compiled extension would address the 17 spent on object creation. Neither was built, because the first question was where the time actually went.

14.8 Exercises

1. Python fails to allocate an output buffer after a transition. Which object must still own the effects, and why is repeating the transition unsafe?
2. An append times out, then its value appears in local history. Which statement would have been false: "the wait ended" or "the command was cancelled"?
3. The memory floor is 100 and the application cursor is 90. Where must commands 91-100 survive, and who is responsible for returning them after restart?

The pending native batch answers the first question: retry the copy, not the transition. The wait ended in the second; cancellation was never established. For the third, the host's durable history must retain the commands independently of the native window and serve them until application and retention obligations have both been met.

14.9 Install and release

Install the Python SDK with `uv add paxodin` or `python -m pip install paxodin`. The wheel contains the native engine; a wheel user needs no Odin compiler. Source builds use the Odin compiler and the core bundled in the source distribution. The separate `paxodin` CLI comes from GitHub releases and works inside a repository checkout. Its build and test commands need Odin; documentation also needs Typst. The Odin import name remains `paxos`.

Tag names are `vMAJOR.MINOR.PATCH`. The release gate checks agreement among the tag, core, CLI and Python versions. Three native jobs build CLI archives and wheels, rebuild wheels from standalone source distributions, and test wheel installs on CPython 3.12, 3.13 and 3.14 without Odin on PATH. macOS releases target only Apple Silicon, with macOS 12 as the minimum. Linux wheel tags encode their glibc requirement. Windows releases target x86-64. Checksums accompany GitHub assets.

Routine CI runs both Odin test profiles, contracts and a short seeded fault matrix, plus Python lint, types and both durability gates. Full fault simulations run on tags, weekly and on manual dispatch. New pushes cancel obsolete branch checks. Build jobs have no publication credentials; only the final publish job receives the organization's `PYPI_API_KEY`. GitHub Pages deploys the built static site with its own narrowly scoped permissions. Performance benchmarks remain reproducible manual experiments rather than noisy shared-runner timing gates.