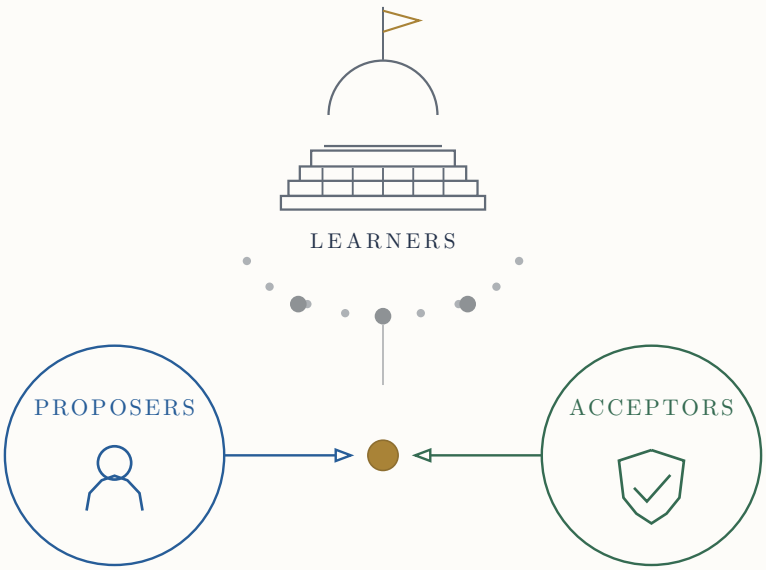
PAXOS

THE PART-TIME PARLIAMENT

A LITERATE GUIDE TO CONSENSUS
FOR UNRELIABLE TIMES



$$|Q| > \frac{N}{2} \quad \Rightarrow \quad Q \cap Q' \neq \varnothing$$

Agreement, even when members come and go.

VIKRANT RATHORE

WITH ASSISTANCE FROM RONAK RATHORE

About this book

This book explains one algorithm. It also explains one implementation of that algorithm. The two tasks are kept together. A line of code is easier to trust when we know the fact that requires it. A proof is easier to remember when we can point to the field that carries its meaning.

The source is written in Typst. The diagrams use Fletcher and CeTZ. The code is Zig 0.16. The protocol is grounded in Leslie Lamport's "The Part-Time Parliament" and "Paxos Made Simple"; the reconfiguration layer is closest to the stop-sign construction described in "Reconfiguring a State Machine".

> *Early in this millennium, the Aegean island of Paxos was a thriving mercantile center.*
>
> - Leslie Lamport, "The Part Time Parliament"

The main promise A careful reader should be able to derive the core Paxos safety rule, trace it into this library's fields and effects, run a three-node example, design the missing host services, and review a deployment without confusing a protocol guarantee with an application guarantee.

Version 0.6.0. Built with Typst 0.15 or later.

1 Preface

The usual introduction to Paxos starts too late. It starts with prepare and accept messages. Those messages then look like rules from a game whose purpose was forgotten.
We shall start earlier. We shall ask a small question.
Three machines must write one word on three pieces of paper. A machine may stop. A messenger may vanish. No machine may erase ink. How can the machines make sure that two different words are never declared final?
The answer will grow in small steps. Each step will remove one bad solution. When prepare and accept finally appear, they will have no mystery left. They are the shortest names for facts that we already need.
The style of this book is mathematical, but it is not terse. We shall compute small examples. We shall predict before seeing answers, explain steps in plain language, and gradually remove hints. We shall make mistakes on purpose. We shall keep a ledger of the facts that survive every mistake.
The implementation follows the same order. First come values and ballots. Next come promises and votes. Then come slots, leader recovery, stable storage, and a replicated state machine. At each point we ask two questions.

1. What can go wrong?
2. Which fact prevents it?

There is one complete runnable example and two progressively larger design studies.

1. The counter on three nodes is `examples/counter.zig`; it exercises the real public API end to end in memory.
2. The key-value service is an explicitly labeled host-design sketch. It adds request deduplication, read semantics, recovery, and snapshot ownership.
3. The regional control plane is an architecture exercise. It adds five voters, sharding, failure domains, metrics, and capacity calculations without pretending that those systems already exist in this repository.

The final parts discuss tests and measurement. The benchmark compares the Zig library with OmniPaxos 0.2.2 in Rust and a pinned LibPaxos3 core in C. The comparison uses the same number of nodes and values. It records protocol differences instead of pretending that the libraries are identical.

> **A word about certainty** Testing can reveal a broken invariant. It cannot create an invariant. We first state the reason that the code is safe. We then use tests to search for errors in our statement and in our code.

1.1 How to read the book

The next chapter explains the learning design and offers a protocol route and a systems-builder route. Parts I through III derive Paxos. Part IV is the library and host-integration manual. Part V moves from the runnable counter to design studies. Part VI covers evidence and operations. Part VII is a desk reference.
Short notes marked "Exercise" are part of the argument. A reader who answers them will remember the proof far longer than a reader who merely agrees with it. Hints appear when a calculation has a trick.
Code fragments are labeled **repository excerpt**, **runnable fragment**, or **design sketch**. Repository excerpts and runnable fragments use the current public API. A sketch may introduce host-owned types such as `Journal` or `Transport`; those types are not library promises.

1.2 Audience and prerequisites

The book assumes that you can read Zig structs, tagged unions, slices, errors, and comptime parameters. It does not assume prior study of Paxos or formal methods. Before building a real service, you should also be comfortable with write-ahead logging, checksums, process crash recovery, authenticated network protocols, and deterministic state machines. The book teaches how those pieces meet this library; it is not a substitute for testing the host implementation.

1.3 Notation

Mark	Meaning
`A1`, `A2`, `A3`	Three acceptors.
`C`	A candidate or proposer.
`L`	A learner.
`b = (r, p, n)`	A ballot with round `r`, priority `p`, and node `n`.
`Q1`, `Q2`	Phase-one read quorum and phase-two write quorum.
`s`	A one based log slot.
`v`	An application value.
`null`	No prior accepted value.

We use the word "node" for a process that may contain all Paxos roles. We use "acceptor" when only its voting duty matters. We use "leader" for a proposer that completed phase one. We use "chosen" for a fact established by a quorum. We use "committed" for a chosen value that a learner has recorded.

1.4 Commands used in the book

```
zig build fmt
zig build test
zig build run
zig build benchmark
zig build book
```

The last command creates `docs/part-time-parliament.pdf`. The aggregate benchmark builds the pinned Rust package and fetches the pinned C source, so its first run needs network access. `zig build benchmark-zig` runs only the local Zig regression workload.

Contents

2 How This Book Teaches

Paxos is difficult for a predictable reason: a learner must hold failures, quorums, ballots, durable state, message order, and application behavior in mind at once. Adding more explanation can make that problem worse. This book therefore reveals one layer at a time and repeatedly returns to one question:

> **The organizing question** What fact prevents two different values from becoming chosen in one slot?

The learning design follows four bodies of work, but does not pretend that they offer the same kind of evidence.

1. **Cognitive load research** motivates small conceptual units, integrated diagrams and prose, worked traces before independent problems, and gradual removal of guidance. Working memory is limited; long-term schemas let an expert treat several interacting facts as one unit.
2. **Self-explanation research** motivates the prompts that ask you to explain why a step is legal, not merely repeat what happened.
3. **Feynman-inspired practice** motivates plain-language teach-backs, concrete analogies, and intellectual honesty at the point where an explanation becomes vague. The popular four-step "Feynman technique" is a later teaching convention, not a controlled method published by Feynman himself.
4. **Knuth's literate-programming principle** motivates presenting code in the order a human needs for understanding. The repository remains ordinary Zig; "literate" here describes the exposition, not a generated WEB program.
5. **Lamport's method** supplies the technical spine: state the goal, derive an invariant, describe state and next-state actions, and only then inspect the message protocol and code.

These principles are design constraints, not guarantees. Learning still requires tracing executions, retrieving ideas without the page, and building something that fails under controlled conditions.

2.1 The three representations

Every important idea appears at three levels. Do not advance merely because the vocabulary looks familiar.

Level	Question	Evidence of understanding
1. Safety goal	What must never happen?	You can state the property without protocol words.
2. State transition	Which state changes preserve it?	You can trace a transition and name the invariant it keeps true.
3. Zig effect	Which field, write, and message implements it?	You can use the public API while preserving write-before-send.

This order follows Lamport's recommended separation between what an algorithm must accomplish, a high-level algorithm, and the message-passing algorithm. It also limits split attention: a code fragment is introduced next to the fact that makes the fragment necessary.

2.2 The learning loop

Each chapter uses the same six moves.

1. **Orient.** Read the learning contract and recall its prerequisites.
2. **Predict.** Commit to an answer before the trace reveals it.
3. **Study a worked case.** Follow every state change and its reason.

4. **Complete a faded case.** Supply one or more omitted decisions.
5. **Teach it back.** Close the book and explain the idea in plain language.
6. **Transfer.** Change the failure schedule, quorum, or application and decide whether the reasoning still holds.

The exercises deliberately fade. Early questions include a hint and name the relevant invariant. Later questions ask you to diagnose an execution or design a host component with less guidance. If you already build consensus systems, start at a chapter's checkpoint and move backward only when your explanation has a gap; excessive guidance can itself become distracting for experts.

2.3 Two routes through the book

Route	Read in order	Do, do not merely read
Protocol learner	Parts I–III, then the counter in Part V, then Part IV.	Draw the quorum traces and answer every teach-back from memory.
Systems builder	This chapter, Parts IV–VI, then return to Parts I–III whenever an invariant is named.	Run the example, implement an effect consumer, and rehearse recovery before adding a transport.

2.4 What this repository actually supplies

The boundary matters. The repository contains:

1. `paxos.Protocol`, a bounded Paxos/Multi-Paxos effect machine;
2. `paxos.ReplicatedLog`, a stop-sign layer for sealed configurations and membership handover on one global slot line;
3. one complete in-memory three-node counter in `examples/counter.zig`;
4. deterministic unit tests for ballots, recovery, duplicates, catch-up, batching, timeouts, reconfiguration, and bounded capacity;
5. local CPU benchmarks whose numbers are regression signals, not production latency claims.

It does **not** contain a production journal, network transport, codec, authentication layer, client session service, snapshot store, linearizable read service, or a general deterministic fault simulator. Parts V and VI show how to design those host components and clearly label non-runnable sketches.

> **Scope is part of correctness** A protocol library can preserve its invariant and the resulting service can still be wrong because the host sent before syncing, reused an identity, applied a command twice, trusted an unauthenticated envelope, or served a read with stronger semantics than it actually implemented.

2.5 Start with retrieval, not recognition

Before Part I, close this page and answer on blank paper:

1. What is the one outcome consensus must forbid?
2. Why might a majority be useful after a crash?
3. Which actions belong to the library, and which belong to the host?

Keep the paper. At the end of Part III, answer again without looking and compare the two explanations. The difference is more useful than a feeling of familiarity.

> **Teach it back.** Explain the book's three levels to a colleague. If your explanation uses the words "Paxos", "ballot", or "quorum" at level 1, try again without them.

PART I

One decision

We begin with one empty line in a ledger. We end with the safety rule that determines every legal Paxos message.

3 Foundations of Consensus

Learning contract By the end of this chapter you should be able to distinguish safety from progress, calculate quorum constraints, order this library's three-component ballots, and explain why intersection without durable memory is insufficient.

Checkpoint: prior knowledge You need only sets, integer arithmetic, and the idea that a process can stop between two instructions. If you already know Paxos, try the teach-back at the end first; return here if you cannot explain why the **highest** accepted ballot matters.

3.1 The empty ledger

Imagine three librarians sitting in separate rooms, each holding a copy of the same ledger. The next line of this ledger is currently blank, waiting for a decision. Now, two visitors arrive at the library. The first visitor sends a messenger to the librarians asking them to write `olive = 7` on that blank line. At the same time, the second visitor sends another messenger asking them to write `olive = 9`.

To make matters worse, the librarians cannot talk directly to each other; they can only send handwritten notes back and forth via messengers. These messengers are notorious for losing notes, falling asleep, or delivering them out of order. However, we assume that when a note does arrive, its content is exact—messengers may lose messages, but they do not alter the words on them.

Our goal is deceptively simple: choose at most one of the proposed values for the blank line, and let learners report a value only after it is chosen. We do not require a decision while every messenger is lost. Agreement and validity are safety properties; eventual choice under additional communication and leadership assumptions is a progress property.

This distinction between safety ("nothing bad ever happens") and liveness ("something good eventually happens") is our first and most important tool.

Safety Nothing bad happens. For Paxos, two different values are never chosen for the same ledger slot. Once a value is chosen, it is locked forever.

Liveness Something good eventually happens. For Paxos, a proposed value is eventually chosen when a quorum of nodes can communicate and one leader remains active long enough to coordinate them.

A stopped system is perfectly safe. It does no work, but it never contradicts itself. This fact lets us design safety rules first, without guessing how long a message might take to cross the network.

Predict before reading on Nodes A and B have durably accepted `olive = 7`, but the leader crashes before telling any learner. Is the value chosen? Write one sentence before continuing. Do not use "the leader knows" in the sentence.

3.1.1 Three tempting answers

If we try to solve this consensus problem, we quickly run into three intuitive but flawed approaches:

1. **The Unanimous Vote ("Ask Everyone")**: We could require all three librarians to agree before writing a value. This works perfectly until one librarian leaves the room or falls ill. Now, the system is permanently stuck because we cannot obtain the final vote.

2. **The Free Majority ("Ask Any Two")**: Since two out of three form a majority, we might allow any two librarians to write a value. But what happens if they vote, restart their machines, and forget their previous votes? A later pair of librarians could meet and write a different value, violating safety.
3. **The Single Leader ("Designate a Dictator")**: We can appoint one librarian as the sole leader who decides all values. But what happens when the leader crashes? How does a newly elected leader safely discover what the old leader might have already decided and committed?

Each of these attempts contains a piece of the final solution. We need a **quorum** of nodes to survive failures, **durable memory** to prevent nodes from forgetting their votes, and a **leader election rule** that respects past decisions.

Exercise 1.1. In a cluster of five nodes, how many nodes form a majority? How many nodes may be completely offline while the system remains capable of making new progress?
Hint: Compute `floor(N / 2) + 1` for the majority size.

3.1.2 The failure model

To build a consensus system that we can prove correct, we must define the rules of the universe in which it operates. We adopt the **Crash-Recovery** model:

1. **Honesty**: A node follows the algorithm exactly while it is running. It does not act maliciously, invent messages, or lie.
2. **Halt**: A node may stop executing at any instruction (a crash).
3. **Recovery**: A node may restart at any time. It recover its state from data written to stable storage (such as a disk journal).
4. **Asynchronous Network**: The network can delay, duplicate, lose, or reorder messages. However, any delivered message is uncorrupted.

If a node could lie or act maliciously, we would be in the **Byzantine** model, which requires more complex protocols and larger quorums. Paxos assumes nodes are correct but unreliable.

The disk is part of the proof A promise that was sent and then forgotten is a lie. If an acceptor replies to a leader before its state is durably written to disk, a sudden power failure can erase its memory, allowing it to violate its promise upon reboot.

3.2 Quorums

How do we make progress when some nodes are dead? We use quorums. A quorum is a subset of nodes large enough that any two quorums must share at least one node. If we use simple majorities, this property is guaranteed mathematically.

Consider three nodes: A, B, and C. The possible majority quorums of size two are: {A, B}, quad {A, C}, quad {B, C} Pick any two of these sets. They **must** intersect. For example, $\{A, B\}$ and $\{B, C\}$ share node B.

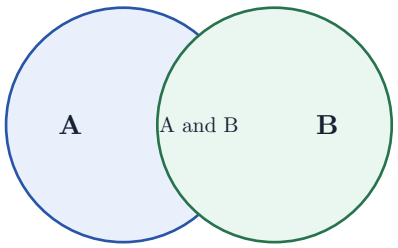


Figure 1: Two majority quorums overlap. The shared node acts as a witness, carrying knowledge of past decisions into the future.

If a value is chosen by one quorum, and we later query another quorum, the overlapping node acts as a **witness**. It carries the memory of the chosen value to the next leader.

3.2.1 Why an odd count is common

In a cluster of four nodes, a majority quorum requires three nodes. The system can tolerate only $4 - 3 = 1$ crash. In a cluster of three nodes, a majority is two, which also tolerates $3 - 2 = 1$ crash. Adding the fourth node increased our costs (hardware, network traffic) without increasing our fault tolerance.

For this reason, voting groups commonly use an odd number of nodes such as three or five. The Zig library uses uniform majorities by default. The compatibility method `Membership.quorum()` returns the phase-two size; the explicit methods are `readQuorum()` and `writeQuorum()`:

```
pub fn quorum(self: *const Membership) usize {
    return self.writeQuorum();
}

pub fn readQuorum(self: *const Membership) usize {
    return self.read_quorum_size;
}

pub fn writeQuorum(self: *const Membership) usize {
    return self.write_quorum_size;
}
```

`Membership.init` computes the majority as `count / 2 + 1` when the options do not supply explicit sizes. The implementation stores validated `read_quorum_size` and `write_quorum_size` fields because options may request flexible quorums. Validation occurs when the actual member count is known.

Since Zig uses integer division, `5 / 2 + 1` evaluates to `3`.

3.2.2 Intersection is not memory

Imagine that quorum $\{A, B\}$ accepts a value x. Later, a candidate contacts quorum $\{B, C\}$. The candidate meets node B, which is the intersection.

But what if node B crashed and rebooted in the meantime, forgetting its vote? The intersection still exists on paper, but the **knowledge** of the decision is gone. Quorum intersection only works if nodes have stable memory.

> **Exercise 2.1.** Construct two quorums of size two in a four-node cluster $\{A, B, C, D\}$ that do not intersect. Why is it impossible to guarantee safety if we define a quorum as any two nodes in a four-node system?

3.3 Ballots

If a leader never crashed, consensus would be trivial. Because leaders fail, we must support a sequence of attempts to make a decision. We call each attempt a **ballot**. Ballots must be unique and totally ordered.

In the Zig library, a ballot is defined as:

```
pub const Ballot = struct {
    round: u64,
    priority: u32 = 0,
    node: NodeId,
};
```

We compare ballots lexicographically: first `round`, then configured `priority`, then `node`. A larger priority cannot overtake a larger round; it only orders candidates in the same round.

Left Ballot	Right Ballot	Comparison Result
`(4, 0, 2)`	`(5, 0, 1)`	Right is greater (higher round dominates).

Left Ballot	Right Ballot	Comparison Result
`(7, 2, 9)`	`(7, 3, 1)`	Right is greater (priority breaks the tie).
`(9, 0, 1)`	`(9, 0, 4)`	Right is greater (node ID breaks the tie).

The node ID ensures that two members do not issue the same complete ballot. If Node 2 uses `(11, 0, 2)` and Node 3 uses `(11, 0, 3)`, the ballots are distinct, and `(11, 0, 3)` is greater. This assumes node IDs are unique and are never reused for a different logical member within the configuration history.

3.3.1 Votes and success

A ballot represents an attempt to choose a value. It consists of:

1. A unique ballot number.
2. A proposed value.
3. A quorum of acceptors.

A ballot **succeeds** (and its value is chosen) when every member of its selected quorum accepts the proposal. The word "chosen" means that a quorum of acceptances exists. The proposer does not need to know that the quorum succeeded for the fact of choice to be true. If the final vote is written to disk, the value is chosen, even if the leader dies immediately afterward.

3.4 Three invariants

Leslie Lamport defined the safety of Paxos using three conditions. Let us translate them into clear programmatic invariants:

Invariant	Rule	Zig Implementation
`B1`	Every ballot number is unique.	The ordered tuple `(round, priority, node)` with a unique member ID.
`B2`	Every phase-one quorum intersects every phase-two quorum.	`Membership.init` validates `read_quorum_size + write_quorum_size > count`.
`B3`	A later ballot must choose the highest-numbered vote already accepted by any node in its phase one quorum.	Phase one promise scanning and value selection logic.

Invariant `B3` is the crown jewel of Paxos. It prevents a new ballot from overwriting or changing a value that might have already been chosen by an earlier ballot.

Suppose Candidate C starts ballot 12. It queries a quorum of acceptors about what they last accepted. The replies are:

Acceptor	Highest Accepted Ballot	Accepted Value
`A1`	`8`	`red`
`A2`	`10`	`blue`
`A3`	`null`	`null`

Candidate C **must** propose `blue` for ballot 12 because `blue` was accepted in ballot 10, which is the highest ballot reported in the quorum. It cannot choose a new client value like `green`.

> **The Selection Rule** If the quorum replies contain no previously accepted values, the candidate is free to propose its own client value. Otherwise, it **must** select the value associated with the highest ballot number reported.

3.4.1 Why the greatest vote matters: a worked proof

Why does selecting the highest ballot preserve safety?

Assume ballot 6 succeeded in choosing `red`. Any later ballot, say ballot 10, must intersect ballot 6's quorum. Therefore, at least one node in ballot 10's quorum will report that it accepted `red` in ballot 6. By following rule `B3`, the leader of ballot 10 is forced to propose `red`.

The careful induction is over later ballots that actually issue proposals. Assume every proposal numbered between 6 and 10 carries `red`. A phase-one quorum for ballot 10 intersects the phase-two quorum that chose ballot 6. Any higher accepted proposal reported by the new quorum is also `red` by the induction hypothesis. Selecting the greatest reported vote therefore selects `red`. Safety is preserved even if nobody yet knows that ballot 6 succeeded.

Exercise 4.1. A phase one quorum reports three accepted values: `(3, apple)`, `(9, apple)`, and `(7, pear)`. Which value must the new candidate propose? Does the answer change if ballot 3 is known to have successfully reached a quorum?

Exercise 4.2. Complete the faded proof. A value `x` was chosen in ballot 12. A proposer for ballot 20 queries a phase-one quorum. The old phase-two quorum and the new phase-one quorum share acceptor `[name]`. That acceptor reports either ballot 12 or a later accepted ballot. By the induction hypothesis, the value of any such later vote is `[value]`. Therefore ballot 20 must propose `[value]`.

Hint: The same value fills the final two blanks.

3.5 From rules to messages

We have derived the protocol simply by thinking through the invariants. To safely propose a value:

1. The candidate needs a ballot number higher than any seen so far.
2. It must obtain a promise from a quorum of acceptors not to accept any lower ballots.
3. It must collect the highest votes those acceptors have already cast.
4. If a prior vote exists, it must use it; otherwise, it proposes its own value.

These four steps map directly to the classic Paxos messages:

In the next chapter, we will follow these messages through their handlers, disk writes, and failure states. We will see how every struct field in our Zig library pays rent by protecting one of these core safety invariants.

Teach it back. Without notes, explain to a new engineer why "ask a majority" is not enough. Your explanation must include a crash, stable memory, an intersecting node, and the rule for selecting one prior vote. Then check whether you said "highest ballot" rather than "most common value".

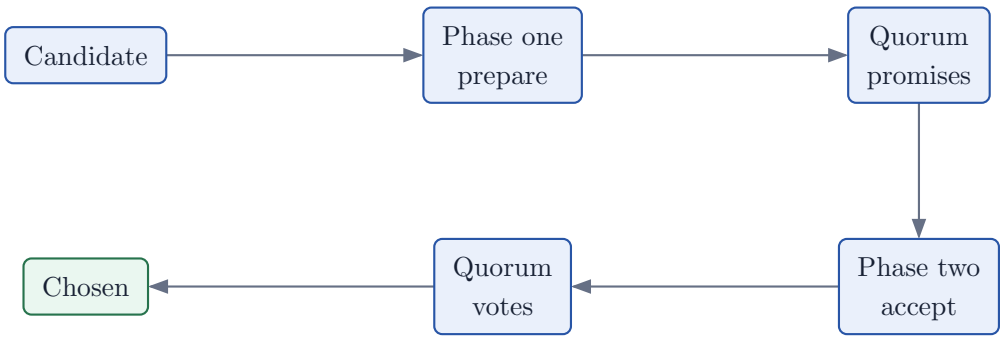


Figure 2: Phase one obtains promises and prior votes. Phase two broadcasts the proposal and collects acceptances to reach consensus.

PART II

The complete ballot

We execute one ballot. We stop at every write, every reply, and every failure. At the end we can recover after a crash without guessing.

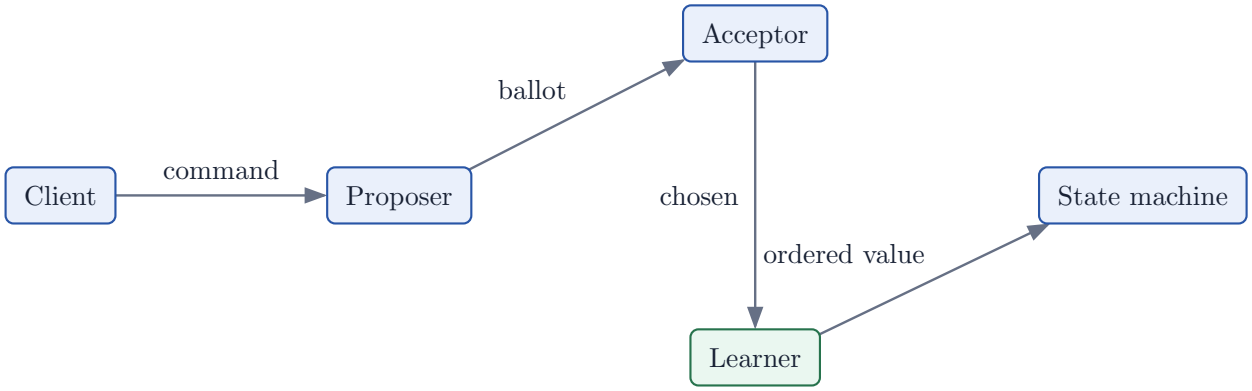


Figure 3: A client supplies the intent (what we want to do). Paxos orders that intent. The state machine executes the ordered commands, turning them into meaningful results.

4 The Single-Decree Protocol

Learning contract By the end of this chapter you should be able to execute one ballot by hand, distinguish accepted, chosen, committed, and applied, place every required disk sync before its dependent message, and predict recovery at any crash point.

Checkpoint: foundation State the highest-vote rule without looking back. If you say "majority value" or "latest message", revisit Part I before learning the message names.

4.1 The four roles

To build a consensus system, we divide the work into four distinct roles: **Proposers**, **Acceptors**, **Learners**, and **Clients**. Although a production server (like a node in our library) usually performs all of these roles at the same time to save resources, they are conceptually completely separate.

Let us break down what each role is thinking and what it must remember:

1. **Clients**: The clients want to get work done. A client submits a command (e.g., `set x = 10`) and expects a response. If the network drops a message, the client must retry. To prevent executing the same command twice, the client attaches a unique `client_id` and `request_id` to its request.
2. **Proposers (Candidates/Leaders)**: Proposers are the active orchestrators. They run campaigns to become the leader, choose values, and drive consensus. A proposer does not need stable storage to ensure safety after a crash; if it restarts, it can simply start a new campaign with a higher ballot.
3. **Acceptors (The Parliament)**: Acceptors are the passive voters. They act as the stable memory of the system. They receive queries, make promises, write votes to disk, and reply to proposers. **Acceptors must never forget what they have written to disk.**
4. **Learners**: Learners are the observers. They do not vote. They listen for chosen decisions and apply them to the local application state machine (such as a key-value store or database).

Role	Key Question	Durable State on Disk
Proposer	Which value am I allowed to propose?	Leadership bookkeeping is volatile; the co-located acceptor still persists promises and votes.
Acceptor	Am I allowed to vote, and what did I vote for?	Highest promised ballot, and highest accepted vote.

Role	Key Question	Durable State on Disk
Learner	Which values have been chosen?	Committed values are durable; this library's delivery cursor is volatile and may replay after restart.
Client	Did my request succeed?	A host-owned stable request identity and result record when exactly-once application effects are required.

4.2 Phase one: The Election and Query

Phase one is the leader election and recovery phase. A candidate cannot propose a value out of thin air. First, it must ask the acceptors what they have already done.

4.2.1 Step 1: Prepare

A candidate chooses a ballot number higher than any it has seen so far (say, `(round = 1, node = 1)`). It sends a `prepare` message to all acceptors:

```
prepare { ballot }
```

Notice that this message contains no value! The candidate is not yet proposing `olive = 7`. It is simply querying the acceptors.

4.2.2 Step 2: Promise

When an acceptor receives a `prepare` message with ballot B, it compares B with the highest ballot it has ever promised ($P_{\max}$).

If $B < P_{\max}$**: The acceptor ignores or rejects the prepare by sending a `nack` (negative acknowledgement). If** $B \geq P_{\max}$: The acceptor makes a solemn promise. It writes B to its durable storage as its new promise ($P_{\max} = B$). Once this write is synced to disk, the acceptor replies with a `promise` message:

```
promise {
    ballot = B,
    accepted_ballot = B_last,
    accepted_value = V_last
}
```

This is the abstract single-decree reply. The library uses Multi-Paxos framing: zero or more `promise { ballot, slot, accepted }` envelopes followed by a `promise_range` descriptor that states which slot range was answered and how many accepted entries it contains. A candidate counts an acceptor only after the stated number of distinct entries has arrived. Part III explains why the separate range descriptor is necessary.

The promise has a powerful meaning: **"I promise never to accept any future proposal that has a ballot number lower than B. Also, here is the highest ballot I have accepted so far (B_{last}), along with the value I voted for (V_{last})."**

> **Sync before you speak** An acceptor must never send a `promise` reply until the promise is written to durable disk. If it replies first, and then crashes before the disk write completes, it might reboot and vote for a lower ballot, breaking the quorum intersection guarantee!

4.2.3 Step 3: Value Selection

The candidate waits for a quorum of complete promise replies. Once it has a quorum, it examines the replies:

1. **If no acceptor has ever accepted a value**: The candidate is free to propose its own client value (e.g., `olive = 7`).
2. **If any acceptor has accepted a value**: The candidate is **forced** to select the value associated with the highest accepted ballot number reported in the replies.

The candidate is now the prepared leader. It can proceed to Phase Two.

> **Predict before reading on** A candidate receives `promise_range` before the corresponding `promise` entry because the transport reordered them. May it count that acceptor toward its phase-one quorum? Write the safety fact that would be lost if it did.

4.3 Phase two: Proposing and Voting

Now that the leader knows what was chosen in the past, it can safely write the future.

4.3.1 Step 4: Accept

The leader broadcasts its proposal to all acceptors:

```
accept { ballot, slot, value }
```

4.3.2 Step 5: Accepted

When an acceptor receives the `accept` message with ballot B and value V, it checks its promise one more time. Why? Because another candidate might have campaigned in the split second between Phase One and Phase Two!

If $B < P_{\max}$**: The acceptor rejects the vote and replies with a `nack`. If** $B \geq P_{\max}$: The acceptor accepts the vote. It writes the accepted ballot and value to disk durably:

$$\text{accepted\_ballot} = B, \quad \text{accepted\_value} = V$$

Once the write is synced, it replies to the leader:

```
accepted { ballot, slot }
```

4.3.3 Step 6: Commit

When the leader collects a quorum of `accepted` votes for its ballot and slot, the value is **chosen**. The leader writes the commit to its local ledger and broadcasts a `commit` message to all nodes (who act as learners):

```
commit { slot, value }
```

Upon receiving `commit`, a node records a commit write and releases only a contiguous prefix through `Effects.committedSlice()`. The host must make the commit record durable before it treats the released application entry as recoverable. A `commit` message is evidence because the non-Byzantine sender is assumed to follow the protocol; `Node.step` is not an authentication layer.

4.4 Local acceptance optimization

A leader node is also an acceptor. In our Zig library, we optimize this by writing the leader's own vote directly to its local disk without sending a network loopback packet to itself:

```
self.durable.promised = self.ballot;
self.durable.accepted[index] = .{
    .ballot = self.ballot,
    .value = value,
};
effects.addWrite(.{ .accept = .{
    .ballot = self.ballot,
    .slot = slot,
    .value = value,
} });
```

This local write counts as the first vote. With the default majority in a three-node configuration, one remote `accepted` reply then completes the phase-two quorum.

4.5 A complete trace

Let us watch this dance in action. Nodes 1, 2, and 3 start empty. Node 1 tries to propose `tea` using ballot `(1, 0, 1)`.

Step	Actor	Event and reason
1	N1	Chooses ballot `(1, 0, 1)` and enqueues `prepare` to all nodes, including itself.
2	N1	Consumes its loopback prepare, writes promise `(1, 0, 1)`, then describes its empty log with `promise_range`.
3	N2	Receives `prepare`, writes promise `(1, 0, 1)`, then replies with no past votes.
4	N1	Collects promises from N1 and N2 (a quorum). No old value was reported.
5	N1	Proposes `tea`. Writes acceptance of `tea` locally to disk.
6	N1	Sends `accept` for `tea` to N2 and N3.
7	N2	Checks promise, writes acceptance of `tea` to disk, and replies `accepted`.
8	N1	Sees two acceptances (N1, N2). `tea` is officially chosen!
9	N1	Writes `commit` to disk and broadcasts `commit` to N2 and N3.
10	All	Learn the commit and feed `tea` to their state machines.

4.6 Rejection and competing campaigns

What happens if Node 2 tries to campaign with ballot `(2, 0, 2)` while Node 1 is leading?
When Node 2 sends `prepare` with ballot `(2, 0, 2)` to Node 1, Node 1 compares it with its own active ballot `(1, 0, 1)`. Since the new ballot is greater, Node 1 records the promise and becomes a follower.
If Node 1 later tries to propose a value, the acceptors will reply with `nack`:

```
nack { rejected = (1, 0, 1), promised = (2, 0, 2), decided_through = 0 }
```

This tells Node 1 that its ballot is stale. The handler records the higher observed round and steps down. A later explicit `campaign` or timeout-driven `tick` chooses a round greater than the local ballot, durable promise, and highest observed round.

4.7 Liveness and the dueling leaders

What if two nodes campaign back and forth endlessly?

- N1 prepares ballot `(1, 0, 1)`. Acceptors promise it.
- N2 prepares ballot `(2, 0, 2)`. Acceptors promise it.
- N1's old accept is rejected.
- N1 prepares ballot `(3, 0, 1)`. Acceptors promise it.
- N2's old accept is rejected.

This is a **liveness hazard**. The library implements deterministic logical timeouts, heartbeats, retransmission, and optional ballot priority, but it does not read a clock or randomize election deadlines. The host controls when each node receives `tick`. A deployment should avoid synchronized campaigns through tick scheduling, configured priority, or an external leadership policy. A lease may help progress or reads, but a lease is a host protocol and must not be treated as part of this library's safety proof.

4.8 Crash points and recovery

Let us inspect what happens if a node crashes at any point in the trace.

Crash Point	Disk State	Recovery Behavior
Before promise write completes	Old promise.	The prepare request is lost. The leader will retry.
After promise write, before reply	New promise.	Leader prepares again. Acceptor replies with its saved promise.
Before accept write completes	No vote.	The proposal is lost. The leader will retry.
After accept write, before reply	Vote is saved.	A future leader's prepare phase will discover this vote.
After quorum, before commit	Votes are saved on quorum.	A future leader is forced to recover and commit this value.

This table shows the elegance of Paxos safety: **no matter where the power cord is pulled, the system recovers to a safe, consistent state.**

Exercise 8.1. Look at Step 8. If Node 1 crashes **before** recording or broadcasting `commit`, has `tea` already been chosen? Can a future leader change slot 1 to `coffee`? Name the durable records that force your answer.

Teach it back. Explain one ballot from the acceptor's point of view. Use only the words "number", "promise", "vote", and "disk" until the final sentence. Then map those four ideas to `Ballot`, `Write.promise`, `Write.accept`, and `DurableState.apply`.

PART III

A sequence of decisions

A service needs more than one chosen value. We arrange decisions in slots, recover an old leader's work, fill holes, and apply one ordered prefix.

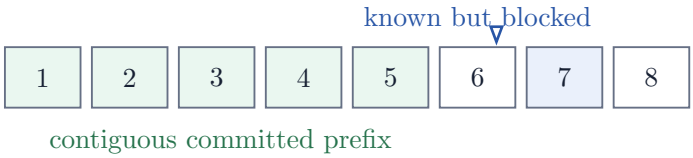


Figure 4: Multi-Paxos runs Phase One once to establish leadership across all slots, allowing subsequent appends to run Phase Two in parallel with a single round trip.

5 Multi-Paxos Log Replication

Learning contract By the end of this chapter you should be able to lift the single-decree rule across slots, explain complete phase-one replies under packet reordering, recover holes with a host-supplied no-op, and describe exactly what a stop sign and the memory floor do—and do not do.

5.1 Why Multi-Paxos?

Classic Paxos chooses exactly one value for one slot. If we want to build a replicated log to run a database, we could run a completely separate instance of Classic Paxos for every single slot in the log.

But this is slow! Every slot would require:

1. Phase One (Prepare & Promise) -> 1 Round Trip.
2. Phase Two (Accept & Accepted) -> 1 Round Trip.

This means every single log append takes at least two network round trips and two disk syncs.

Multi-Paxos is an elegant optimization. Instead of running Phase One for each slot, a candidate campaigns for **every unresolved slot** at the same time. Once the candidate wins a quorum of promises covering them, it becomes the stable leader.

For all subsequent slots, the leader can skip Phase One entirely and propose values in Phase Two directly! The cost of a log append drops to a single network round trip.

5.2 Combining the Phase One Replies

How does a candidate query a log whose slots never end? It cannot ask for "everything": slots are 64-bit and never reset, so a complete answer could be unbounded. Phase one therefore runs in chunks. The candidate sends `prepare (ballot, first)`, naming the first slot it wants resolved. Each acceptor replies with its accepted votes inside that chunk, then describes the chunk it answered:

```
promise (ballot, slot 101, accepted_ballot 3, value "A")
promise (ballot, slot 103, accepted_ballot 4, value "C")
promise_range (ballot, first 101, last 164, accepted_count 2, more false)
```

Because the network can reorder packets, the `promise_range` descriptor might arrive before the individual slot `promise` entries. If the leader immediately declared itself ready, it might miss some accepted entries, violating safety!

To prevent this, the candidate tracks the expected entry count from `promise_range` and waits until it has received every single entry:

```
// Equivalent to Protocol.Node.maybeResolveChunk.
if (!peer.range_described) continue;
if (peer.received_in_range >= peer.expected_in_range) {
    complete += 1;
}
```

A member's reply is counted toward the quorum only when it is complete. This count-based tracking makes the protocol transport-independent: we do not assume TCP FIFO ordering for correctness.

When a counted member reports `more`, the candidate resolves the current chunk and prepares again at the next one. A chunk holds at most `recovery_chunk_slots` slots, so every recovery message and buffer is bounded by the chunk, never by history. The descriptor also carries two fences: `chosen_through`, the acceptor's contiguous chosen prefix, and `anchor`, its adopted trim anchor. The elected leader takes the greatest fence reported by the quorum and never fills, proposes, or accepts a client value at or below it. A missing vote down there means the slot was released, not that it is open.

5.3 Holes and the No-Op Value

Imagine that N1 becomes the leader. During its Phase One recovery, it discovers:

- Slot 1 has an accepted vote `alpha`.
- Slot 3 has an accepted vote `gamma`.
- Slot 2 has no accepted votes reported by anyone.

By the safety rules, the leader must recover and propose `alpha` in Slot 1 and `gamma` in Slot 3. But what about Slot 2? The leader cannot leave Slot 2 empty. If it did, and later applied Slot 3, the database would have a gap in its history, violating state machine order.

The leader must fill the gap in Slot 2. It proposes a **No-Op** (no-operation) command. A No-Op command consumes the slot, but when the state machine applies it, it performs no work.

In our Zig library, the host application supplies the No-Op value when starting a campaign:

```
try node.campaign(.{
    .client_id = 0,
    .request_id = 0,
    .operation = .noop,
}, &effects);
```

This keeps the library clean and generic: the consensus core does not need to invent application-specific command values.

> **A no-op is still a real value** It must be self-contained, serializable, deterministic when applied, and safe to replay. `campaign` stores the supplied value for recovery; there is no special no-op tag inside the generic protocol.

5.4 The Stable Leader Pipeline

A stable leader can have multiple proposals in flight at the same time. This is called **pipelining**. It hides network latency by allowing the leader to propose Slot 101 before Slot 100 has committed.

However, pipelining introduces the risk of unbounded memory usage. If clients send writes faster than the disk can sync them, the queue of uncommitted proposals will grow forever.

The library prevents unbounded protocol memory with a slot-tagged consensus window of `window_slots` physical cells, a compile-time power of two. Slot `s` lives in cell `s & (window_slots - 1)`, tagged with its slot number, so a reused cell can never be mistaken for an earlier occupant. A cell is reused only after the host licenses it: `advanceMemoryFloor(through)` records that every released entry through `through` has been durably consumed. When `next_slot - memory_floor` would exceed the window, `propose` returns `error.WindowFull`. That is transient flow control, not a log limit: retry after the floor advances. The host should still impose an earlier in-flight limit and apply client backpressure before the window fills.

5.5 Membership Changes: The Stop Sign

A consensus cluster cannot remain fixed forever. Machines wear out, datacenters change, and operators must add or remove nodes.

If we simply change the membership configuration on the fly, we risk splitting the cluster. For example, if we transition from three nodes $\{A, B, C\}$ to a new set $\{D, E, F\}$, a partition could allow $\{A, B\}$ to make decisions under the old configuration, while $\{D, E\}$ make different decisions under the new configuration.

Our library implements a safe, clean reconfiguration mechanism called a **Stop Sign**:

> **Stop Sign** A special log entry that names the next configuration. The proposer seals local appends as soon as `reconfigure` succeeds; another node seals after it accepts or commits the stop. Once the stop is decided, the host may transfer state and initialize the next configuration.

```
const slot = try node.reconfigure(
    next_configuration_id,
    &.{ 2, 3, 4, 5, 6 }, // New membership IDs
    "registry_digest",
    &effects,
);
```

By placing the stop inside the log, the old configuration agrees on the boundary relative to commands. The library does not automatically transfer application state, start processes, or stop old network traffic. The host must wait for `isReconfigured()` to return the **decided** stop before calling `initFromStop`, and must prevent the sealed old instance from serving new writes. The next configuration continues the same global slot line at the stop slot; slot numbering never restarts.

5.6 Global Slots on One Line

Slots are 64-bit and global: they never reset for the lifetime of the log. There is no "log full" event and no snapshot rollover. What the window bounds is residency, not history: at most `window_slots` slots live in protocol memory, and everything below the memory floor survives only in the host's journal and materialized state. Three host duties follow.

1. **Advance the floor**: After durably consuming released entries, call `advanceMemoryFloor` so the window keeps moving. A stalled floor eventually turns every proposal into `error.WindowFull`.
2. **Adopt chosen trims**: When the cluster chooses a trim record, call `installChosenTrim` with its anchor. A trimmed acceptor then answers phase one for the released prefix from the anchor instead of from cells it no longer holds.
3. **Serve old history**: A peer catching up from below the memory floor cannot be answered from the window. The core emits a `serve_range` host request, and the host replies with commit envelopes read from its own journal.

An accepted-but-unchosen slot is never evicted, so no vote can be silently forgotten. The only terminal condition is `error.GlobalSlotExhausted` at the end of the 64-bit slot space; at one million commits per second that takes more than 584,000 years.

A stop sign still seals a configuration, but only for membership change. The next configuration continues the same slot line: `initFromStop` starts it at the stop slot with the inherited trim anchor, and no slot number is ever used twice.

Exercise 11.1. A leader has delivered through slot 80 and holds accepted but undecided commands in slots 81 and 82. The window has 64 cells and the host has advanced the memory floor only to slot 40. How many new slots can the leader propose before `WindowFull`? What must the host do to free more, and why can the cells holding slots 81 and 82 never be reclaimed to do it?

Teach it back. On blank paper draw one row of eight window cells and a longer slot line above it. Mark the memory floor, the delivered prefix, and one accepted but undecided slot. Explain which cells may be retagged for later slots, what licenses that, and why the accepted cell must stay.

PART IV

The Zig library

The proof becomes a bounded state machine. The host owns disk, network, time, and application state. The boundary between them is the main API.

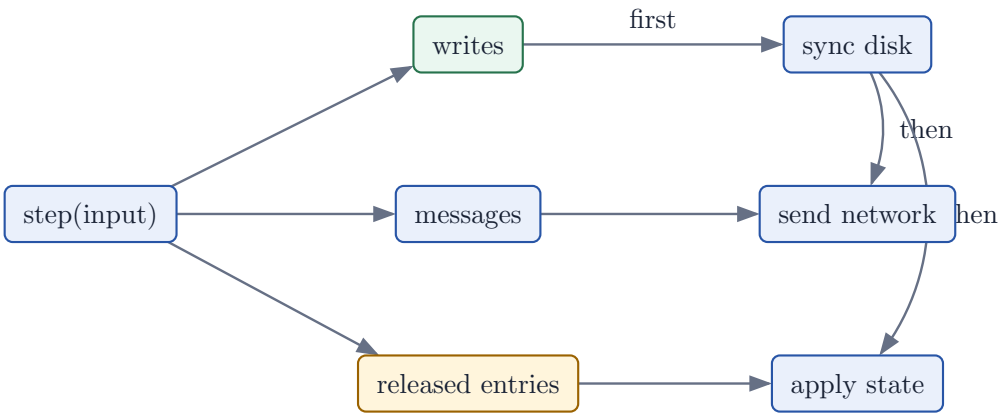


Figure 5: The deterministic consensus core receives input events and outputs effects. The host application owns the actual execution of disk and network writes.

6 Bounded Core State Machine

Learning contract By the end of this chapter you should be able to instantiate `Protocol`, run each public transition, drain one `Effects` batch safely, replay durable records, and identify every service the host must provide.

6.1 Design Philosophy: Consensus Without I/O

This library deliberately excludes sockets, threads, filesystem calls, codecs, and wall-clock reads. The protocol is a deterministic, **mutating** effect machine: a call changes `Node` immediately and describes required external actions in caller-owned `Effects`. It is not a pure function and its in-memory state is not a substitute for the returned durable writes.

Instead, the library acts as a state engine that takes an input event, updates its internal state, and writes its decisions to a caller-supplied **Effect Buffer**.

This design gives the host application complete freedom:

1. The same core can be hosted by TCP, QUIC, a thread pool, or a simulator.
2. Tests can control message delivery and reconstruct nodes from selected durable writes. The repository's unit-test harness is deterministic; it is not yet a general seeded crash simulator.
3. The critical "persist before durable claim" invariant is explicit at the host call site; the core separately classifies the narrow request-only class a host may pipeline while its barrier runs.

6.2 Defining the Protocol Configuration

To use the library, you define your state machine Command type and set compile-time bounds:

```
const paxos = @import("paxos");

// Your application command
const Command = struct {
    client_id: u128,
    request_id: u64,
    operation: enum { put, delete },
    key: [32]u8,
    value_hash: [32]u8,
};

// Instantiate the protocol module
```

```
const Consensus = paxos.Protocol(Command, .{
    .max_members = 5,
    .window_slots = 16_384,
});
```

6.2.1 Comptime Pointer Protection

A critical safety invariant is that message payloads must be self-contained. If `Command` contained a pointer (e.g., `[]const u8`), that pointer would only point to memory in the sender's address space. It would be meaningless to a remote peer receiving the message over the network.

To protect you from this class of bugs, our library uses **comptime metaprogramming reflection**. When you instantiate `paxos.Protocol(Command, ...)`:

- The library scans the `Command` struct at compile time.
- If it detects any pointers, slices, or references, it halts compilation with a clear error: `"Value type '...' must not contain pointers, slices, or references."`
- This forces the protocol value to own its in-memory bytes. The host must still define a versioned, canonical wire and journal encoding and validate decoded lengths and enum tags before constructing an envelope.

Invalid capacity and timer options fail the same way. A zero or non-power-of-two `window_slots`, a `recovery_chunk_slots` above the window, a member bound that does not fit its wire type, a zero tick interval, or a derived effect capacity that would overflow `usize` each stop compilation with a named message such as `paxos Protocol option window_slots must be a power of two`. The `ReplicatedLog` and `Learner` factories validate their options with the same mechanism. Quorum sizes stay a runtime check because the member slice is supplied to `Membership.init` at runtime.

> **API anchor: `paxos.Protocol(Value, options)`** Generates a concrete family of message, durable-state, effect, membership, and node types. A message or write from a differently configured family is not wire-compatible merely because its Zig fields look similar. in `src/protocol.zig`

6.3 The Effect Contract

When you feed an event into the node, you pass a reference to an `Effects` buffer. The node writes its actions to this buffer. **You must execute these effects in a strict order**:

```
// 1. Deliver the envelope to the node
try node.step(envelope, &effects);

// 2. Write and sync to disk journal FIRST
for (effects.writesSlice()) |write| {
    try journal.append(write);
}
try journal.sync(); // Blocking sync

// 3. Record that the batch is durable
effects.confirmWritesDurable();

// 4. Send network messages SECOND
for (effects.messagesSlice()) |message| {
    try transport.send(message);
}

// 5. Apply released entries in slot order.
for (effects.committedSlice()) |entry| {
    try db.apply(entry.slot, entry.value);
}
```

All writes in one batch must be appended in slice order and made durable before **any** message from that batch leaves. Do not invoke another node transition until the batch has been consumed: every public transition resets the active effect counts and may overwrite the backing arrays. Application delivery can follow the sends, as above, but the host must atomically persist its state machine mutation and applied-slot cursor if it needs exactly-once effects.

One more duty closes the loop. Slots are 64-bit and global, and protocol memory is a fixed window of `window_slots` tagged cells. Once the host has durably consumed released entries, it calls `node.advanceMemoryFloor(through)` to license cell reuse below that slot. A host that never advances the floor eventually sees `error.WindowFull` on every proposal; that is backpressure, not a log limit.

The library enforces this order in every optimize mode, including `ReleaseFast` and `ReleaseSmall`. Reading `messagesSlice` before `confirmWritesDurable`, or resetting a batch whose writes were never confirmed, stops the process with a stable diagnostic on stderr:

```
paxos: messagesSlice before confirmWritesDurable
paxos: reset discarded unconfirmed writes
```

A stopped node costs availability; a node that forgets a promise or vote it already published can cost agreement. Operators should treat either line as a host integration bug, preserve the journal, and restart only after the bug is fixed.

6.3.1 The host-managed exception

A host that batches several transitions behind one shared storage barrier copies writes and messages into its own buffers and syncs once for the group. Such a host declares its protocol through `paxos.host_managed.Protocol`. The factory accepts the same options but omits the runtime ordering check, which transfers the safety obligation to the host: copy effects out before the next transition resets them, keep pending messages private until the barrier completes, release nothing after a failed append or sync, and prove with a crash-recovery test that the written prefix rebuilds the live durable state. The namespace is deliberately searchable; every use is an audited exception with written ownership notes. The `ReplicatedLog` layer always builds on the enforced factory and has no host-managed variant.

> **A failed sync invalidates the live node** The call has already mutated `Node`. If append or sync fails, stop using that in-memory node. Terminate or discard it, repair storage, replay only verified durable records into a fresh `DurableState`, restore a fresh node, and repair application state independently.

6.4 The public transition surface

Operation	Host event and result
`init`, `restore`, `restoreAt`	Bootstrap empty state or reconstruct protocol state from replayed `DurableState`, optionally resuming at the host's consumed floor.
`campaign(noop)`	Start phase one explicitly.
`tick(noop)`	Advance logical failure detection, heartbeat, and resend counters by one host-defined interval.
`step(envelope)`	Process one already decoded, validated, authenticated, correctly routed member message.
`propose`, `proposeBatch`	Assign one or more slots on a prepared leader.
`reconnected`	Ask the current role to repair one restored peer link.
`requestCatchUp`	Emit `learn` for a nonzero starting slot.

Operation	Host event and result
`committedAt`, `readDecided`	Inspect learned values; these are log-state reads, not linearizable application reads.
`currentLeader`, `decidedThrough`	Return diagnostic hints and the released contiguous prefix.
`advanceMemoryFloor`, `memoryFloor`	License consensus-cell reuse below the durably consumed prefix; read the current floor.
`installChosenTrim`, `trimAnchor`, `beginRecovery`	Adopt a chosen trim anchor, read it back, or reset onto an installed state image at an anchor.

Every operation can return an error before producing a useful batch. Initialize an `Effects` value once with `effects.init()`, then let each transition reset its counts. Use `paxos.explainError(err)` for an operator-facing explanation; do not treat every error as retryable.

6.5 Metaprogramming and Bounds Optimization

We want the library to support thousands of slots without wasting CPU cycles. To solve this, we optimized our custom `BitSet` implementation:

```
// The hybrid BitSet adapts to the size at compile time
pub fn BitSet(comptime size: usize) type {
    if (size <= 64) {
        return struct {
            bits: std.meta.Int(.unsigned, size) = 0,
            // Inline bit shifting...
        };
    } else {
        const word_count = (size + 63) / 64;
        return struct {
            words: [word_count]u64 = .{0} ** word_count,
            // Array word shifting...
        };
    }
}
```

For at most 64 elements the backing integer has exactly the requested bit width—`BitSet(7)` occupies one byte in the current test. Larger sets use a fixed array of `u64` words. This is an implementation detail worth measuring, not a promise that a particular compiler will choose one instruction.

6.6 Restoring State After a Crash

When a node restarts, it does not keep any volatile leadership or voter memory. It reconstructs its stable consensus state simply by replaying its journal:

```
var durable: Consensus.DurableState = .{};

// Replay records from disk
for (records) |record| {
    try durable.apply(record);
}

var node: Consensus.Node = undefined;
try node.restore(my_node_id, &membership, &durable);
```

Upon restore, the node is a follower. A later heartbeat can identify a leader; enough `tick` calls can start a campaign. Phase one recovers accepted protocol values from a complete read quorum.

`restore` does not restore your database, client sessions, state images, transport queues, or authentication state. It also resets the volatile delivery cursor: `decidedThrough()` reports zero until this node next

observes a commit or wins an election. A host that has already durably consumed a prefix uses `restoreAt(id, membership, durable, floor, priority)` instead; the memory floor and delivery cursor resume at that floor, which keeps a replayed window whose early cells were reused deliverable. Restore the application state and its applied slot from the host's own durable data; if that cursor trails committed journal records, replay those values idempotently before serving traffic. Do not wait for `committedSlice()` to reconstruct the entire old application state.

`DurableState.apply` replays one configuration's journal strictly: a regressed promise or conflicting value is corruption and stops the node. A journal that spans configuration changes legitimately contains ballots from older lines, so lifetime replay uses `DurableState.replayFold` instead: promises fold to their maximum, accepts whose cells were reused by newer slots are skipped as dead history, and commits and trim anchors keep their strict rules.

Teach it back. Draw a vertical boundary labeled library/host. Place `Node`, `Effects`, journal bytes, socket bytes, tick scheduling, application state, and snapshot files on the correct side. Add one arrow that is forbidden until an `fsync` succeeds.

7 Advanced Replicated Log Features

> **Learning contract** By the end of this chapter you should be able to tune logical timers without confusing them with safety, validate flexible quorum arithmetic, repair a lagging follower, and use `ReplicatedLog` without attributing host-owned journal and state-transfer work to the library.

7.1 Logical Time: The Power of Ticks

Paxos safety does not depend on accurate clocks. Progress mechanisms do need a way to suspect silence and retransmit. Reading the system clock inside the core would make identical input schedules harder to reproduce, so the host converts elapsed time or scheduler events into logical ticks.

Our library keeps time out of the consensus core. The node has no clock. Instead, the host application drives time by calling the `tick` method at a stable interval:

```
// Call this every 10 milliseconds in your event loop
try node.tick(noop_command, &effects);
```

When you call `tick`, the node performs three logical duties:

1. **Liveness check (Follower)**: Counts ticks since accepted leader traffic. At `election_timeout_ticks` it starts a campaign; this is suspicion, not proof that the prior leader failed.
2. **Heartbeat emission (Leader)**: Sends heartbeats at `heartbeat_interval_ticks` to remind followers that it is active, preventing split campaigns.
3. **Retransmission (Leader)**: At `resend_interval_ticks`, the leader sweeps its slot log and resends uncommitted proposals to slow or returning peers.

Because ticks are input calls, a test or simulator can advance logical time by calling `tick` without waiting for real milliseconds. This makes timeout paths reproducible; it does not make all liveness schedules trivial to test.

7.2 Flexible Quorums: Shifting the Balance

In classic Paxos, majorities are used for both Phase One (prepare) and Phase Two (propose). For a five-node cluster, both quorums must be size 3.

Howard, Malkhi, and Spiegelman made the Flexible Paxos trade-off explicit: every possible phase-one quorum must intersect every possible phase-two quorum. The current library supports uniform quorum **sizes**, for which the condition is: $|Q_1| + |Q_2| > N$ Two phase-two quorums need not intersect. Within

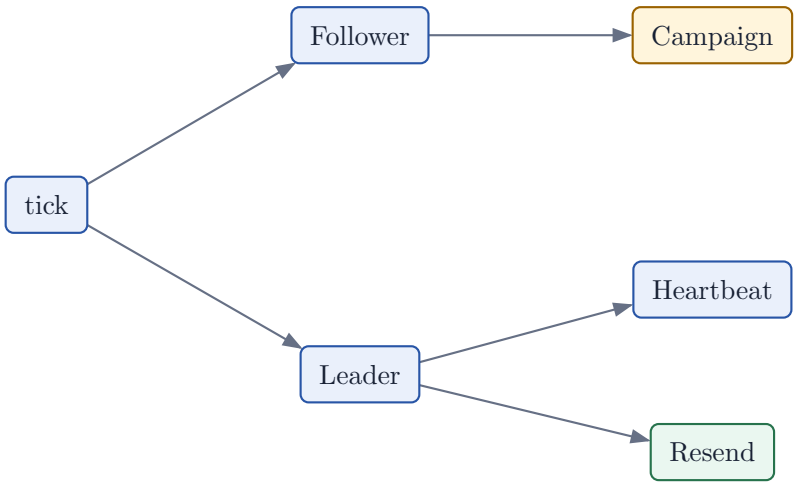


Figure 6: One logical tick drives election timers, heartbeats, and retransmission sweeps. Only the logic for the node's current role is executed.

one ballot, uniqueness and the single-value check prevent two values; before a later ballot proposes, its phase-one quorum intersects the earlier phase-two quorum and recovers the highest vote. Flexible quorums change availability: a five-node `(Q1=4, Q2=2)` system can write with two members under a stable leader but needs four members to replace that leader.
This allows you to tune your cluster for different workloads:

Nodes (N)	Read (Phase 1)	Write (Phase 2)	Operational Trade-off
5	3	3	Standard symmetric majority quorum.
5	4	2	Smaller stable-leader vote quorum; leader replacement needs four members.
5	2	4	Smaller phase-one quorum; every new value needs four durable acceptances.

The options are compile-time constants, but validation occurs when `Membership.init` receives the runtime member slice:

```
const P = paxos.Protocol(Command, .{
    .max_members = 5,
    .window_slots = 8192,
    .read_quorum_size = 4, // Phase 1
    .write_quorum_size = 2, // Phase 2
});
```

7.3 Reconfiguration: The Stop Sign Invariant

When we need to change membership (e.g., adding node 6), we must not let two different configurations run at the same slot.
We solve this using a **Stop Sign** entry. A Stop Sign is a special command proposed in the log:

```
const next_members = [_]NodeId{ 2, 3, 4, 5, 6 };
const slot = try node.reconfigure(
    next_configuration_id,
    &next_members,
    "registry_digest",
    &effects,
);
```

When a stop sign enters the log:

1. **Immediate local seal**: The proposer seals when `reconfigure` returns; another member seals after accepting the stop. This conservative seal can later be cleared by recovery if a higher ballot chose a command instead.
2. **Order Conservation**: It continues processing network messages to help decide and commit all slots up to the Stop Sign.
3. **Clean handover**: Once `isReconfigured()` returns the decided stop, the host calls `initFromStop` with the stop, its slot, and the current trim anchor. The new configuration continues the same global slot line at the slot after the stop; slots do not restart.

Placing the stop inside the log ensures that configuration changes are totally ordered alongside regular writes, preserving safety.

7.4 Catch-Up and Reconciliation

If a node gets partitioned and falls behind, it does not need to run a complex recovery loop. It simply asks its peers for missing commits:

```
try node.requestCatchUp(peer_id, first_missing_slot, &effects);
```

The request is chunk-bounded: the peer replies with the committed entries it still holds in its window, at most `recovery_chunk_slots` per exchange, and the lagging node asks again as its prefix advances. It may

receive entries out of order, but releases only a contiguous prefix. History below the peer's memory floor has left the window entirely; for that range the peer's core emits a `serve_range` host request, and the peer's host replies with commit envelopes read from its journal. Only a gap below the cluster's chosen trim anchor, where journal bytes may be deleted, requires the host to install a verified state image at an anchor (`beginRecovery`) and replay the retained suffix.

Exercise 14.1. For seven members, choose uniform phase-one and phase-two sizes optimized for a small stable-leader write quorum of three. What phase-one size is required? How many unavailable members can the cluster tolerate during leader replacement? Check the exact inequality used by `Membership.init`.

Teach it back. Explain why `read_quorum_size` is not an application read API. Then explain one safe way to serve a linearizable key-value read using an ordered barrier, without claiming that `committedAt` itself provides linearizability.

8 Writing Reviewable Consensus Code

Learning contract By the end of this chapter you should be able to review a change in reasoning order, distinguish recoverable input errors from internal invariant checks, and connect a source transition to the proof obligation it preserves.

8.1 Why Zero Allocation Matters

In typical application development, we allocate memory on the heap whenever we need a new buffer or object. If we run out of memory, the operating system might kill our process, or the program might crash. Dynamic allocation is not inherently a Paxos safety violation: a node that fails cleanly on allocation failure can remain fail-stop. It does introduce additional failure paths, latency variance, fragmentation, and ownership work. A torn journal record is a storage-format and recovery problem whether or not a heap was involved. This library chooses fixed bounds so its core never needs to solve those problems at runtime.

Our library follows the **TigerStyle** coding guidelines:

1. **Zero runtime allocation**: All memory needed by the consensus core is allocated statically at startup or on the call stack. The library does not import or use a heap allocator.
2. **Static bounds**: Cluster size, slot log capacity, and message buffers are defined at compile time.
3. **Fail-fast checks**: Recoverable boundaries return errors; safe builds also assert internal invariants close to their mutation sites.

It also follows a Knuth-inspired literate rule: organize an explanation around the human proof, not the compiler's file order. That does not mean comments should narrate syntax. A useful source comment names the invariant, the reason for a write, or the ownership boundary that a reviewer could otherwise miss.

Lamport's hierarchical-proof practice adds another rule: a long argument needs levels. First state the transition's claim; then split it into obligations such as membership validation, durable monotonicity, quorum evidence, and emitted effects. A reviewer should not have to remember an unstructured page of prose while inspecting one branch.

8.2 Control Flow Rules

Consensus transitions should be obvious. We enforce the following control flow guidelines:

- Keep control flow explicit and shallow. Do not hide protocol transitions in callbacks, reflection, or generic magic.
- Return early when a precondition fails to keep the happy path flat.
- Split compound conditions when their parts protect different invariants.
- Do not use recursion in protocol or storage paths.
- Bound every loop with a compile-time capacity or a validated input length.
- Use one explicit, flat switch statement for message dispatch.

8.3 Memory and Type Guidelines

- The consensus library does not allocate after initialization.
- Capacities are compile-time options and are validated at startup.
- Initialize large values through destination pointers to avoid stack copy overhead.
- Do not return a large node struct by value.
- Use fixed-width integers (`u8`, `u32`, `u64`, `u128`) for protocol identities, rounds, slots, and counts.
- Use the native `usize` type **only** for Zig array and slice indexing.

- Values stored in messages must own their data or contain durable identifiers.

8.4 State and Invariant Safety

- State the invariant in code comments before implementing its state transition.
- Assert important preconditions and postconditions in safe builds.
- Pair an assertion before and after a buffer count mutation.
- Durable state (promised ballot, accepted ballot, committed slot) never moves backward.
- One ballot and slot must never carry two conflicting values.
- A committed slot must never change its value.
- A later leader preserves the highest accepted value reported by the phase one quorum.
- Quorum configuration is valid only when read and write quorums intersect.
- Every message that depends on a write is sent **only** after that write is durably synced to disk.

8.5 Functions and Names

- A function should do exactly one protocol action.
- Keep functions at or below 70 lines. Generic factories are exempt, and `tools/check-style.awk` currently also exempts a fixed list of large host modules whose oversized functions are recorded debt, not policy.
- Keep source lines at or below 100 columns.
- Keep every file at or below 2,800 lines of code, counting neither comment-only lines nor blanks. Two host modules exceed this today and are pinned in the checker at their current size, so they can only shrink; new files never get a pin.
- Use names directly from the protocol: `promised`, `accepted`, `committed`, and `ballot`.
- Include units or domains in names when confusion is possible.
- Avoid abbreviations except for established protocol terms.
- Place helper functions immediately after the public operation that motivates them.

8.6 Errors vs Assertions: Knowing the Difference

We distinguish between two kinds of failures:

1. **Operating Errors**: These are expected events in an asynchronous network. A client sends a message to the wrong node, or proposes a command when the log is full. These are handled gracefully by returning an error or a network rejection (`nack`).
   - We explain all operating errors using a human-friendly diagnostics format, inspired by the Elm language compiler's diagnostic design (Compiler Errors for Humans).
   - Each error returns a structured explanation with a clear top boundary header (e.g. `-- NOT LEADER --`), a simple English description of what failed in the state machine, and a `Hint:` suggesting a resolution path.
2. **Invariant Violations**: These are bugs. An acceptor votes for a ballot lower than its promise, or the write count exceeds the size of the array. These represent impossible state transitions.

Internal impossible states use assertions in builds where runtime safety is enabled. Untrusted bytes, wrong recipients, stale ballots, capacity limits, and storage failures must use validation, errors, or protocol replies instead; correctness must not depend on a debug assertion that an optimized build may omit.

A third category sits between them: a **host-facing safety boundary**, where the caller's code, not the library's, can break a load-bearing rule. The effect ordering contract is the canonical example. Boundary checks like this must be always-on, in every optimize mode, and stop the process with a stable diagnostic; an invalid option value at a type factory must fail compilation outright. A boundary is not an internal invariant, so `std.debug.assert` is the wrong tool for it, and it is not an operating error, so returning an error a caller could ignore is wrong too.

In consensus engineering, a dead node is safe, but a corrupt node that continues running and sends corrupt messages can break safety for the entire cluster. We use `std.debug.assert` to protect all internal boundaries:

```
// Verify the array index is within bounds before mutating
std.debug.assert(self.messages_count < self.messages.len);
self.messages[self.messages_count] = envelope;
self.messages_count += 1;
std.debug.assert(self.messages_count <= self.messages.len);
```

8.7 Comments and Documentation

- Comments explain **why** a rule exists, not what the next statement says.
- Public methods document ordering, durability, bounds, and memory ownership.
- Examples must show the write-before-send sequence explicitly.
- Documentation must distinguish safety from liveness.
- Unsupported behavior must be stated directly in documentation, not hidden behind broad claims.

8.8 Tests and Measurements

- Every protocol bug receives a deterministic failure schedule test to prevent regressions.
- Test duplicates, packet loss, reordering, restarts, stale ballots, and bounded capacity.
- Test one voter, the smallest quorum, and the configured maximum.
- A benchmark validates its checksum and protocol message count.
- Cross-language measurements must pin dependency versions and disclose differences.
- Performance claims use observed numbers and never infer language superiority.

8.9 Review Checklist: A Human Proof Outline

Human review complements executable tests; this repository does not yet ship a machine-checked specification that refines to the Zig code. When reviewing a change to the consensus core, ask these questions in order:

1. **What invariant does this change affect?** Trace the code back to Leslie Lamport's invariants `B1`, `B2`, or `B3`.
2. **Is the write synced before the send?** Ensure no network message leaves before the corresponding write is durable.
3. **What happens if this message is duplicated or delayed?** Verify that the handler is idempotent.
4. **Are all buffers bounded?** Make sure no loop or array mutation can run out of bounds.
5. **Is it readable?** Can a colleague review the code and understand the proof without reading this book?

Teach it back. Choose `Node.onAccept` or `DurableState.apply`. Explain it in reasoning order: precondition, state protected, durable change, emitted evidence, duplicate behavior, and failure behavior. Only then read the function and list any line your explanation failed to predict.

PART V

Three worked systems

We run the repository's counter, complete a key-value host design, and review a regional control-plane architecture. Guidance fades as the systems grow.

9 Three Worked Systems

Learning contract By the end of this part you should be able to trace the runnable counter's effect loop, add durable request deduplication and explicit read semantics, and decompose a larger service into independently recoverable shards without inventing guarantees the library does not provide.

9.1 Small Example: The Replicated Counter

Status: complete runnable repository example. Run it with `zig build run`.

The command in `examples/counter.zig` is self-contained:

```
const Command = struct {
    client: u32,
    request: u32,
    operation: enum { noop, add },
    amount: i64,
};

const P = paxos.Protocol(Command, .{
    .max_members = 3,
    .window_slots = 32,
});
```

The example initializes three nodes, campaigns node 1 with a no-op, proposes three `add 10` commands, and drains every generated envelope. The central function is not `propose`; it is the effect consumer:

```
// Repository excerpt: examples/counter.zig
for (effects.writesSlice()) |write|
    try disks[node_index].apply(write);

for (effects.messagesSlice()) |message| {
    queue[queue_count.*] = message;
    queue_count.* += 1;
}

for (effects.committedSlice()) |entry| {
    if (entry.value.operation == .add)
        counters[node_index] += entry.value.amount;
}
```

All three counters reach 30. `DurableState` and the queue are memory in this demonstration. The ordering is real; durability and transport are simulated. The `client` and `request` fields are unique in the workload, but the demo does not deduplicate them.

Exercise 16.1. Change only the in-memory delivery schedule so the final queued envelope is delivered first. Predict the counters before running. Next deliver one envelope twice. Which library test gives confidence about protocol idempotence, and what application behavior is still untested?

9.2 Middle Example: A Key-Value Host

Status: design sketch. `BoundedMap`, `Result`, journal, codec, snapshot store, and client service are not supplied by this repository.

A key-value service may order hashes that reference separate immutable blob storage. It also has to resolve client retry ambiguity: a write may commit even when its reply is lost.

9.2.1 A bounded request discipline

One compact design permits at most one outstanding request per client and persists the most recent ID and result with the application state:

```
const ClientRecord = struct {
    request_id: u64,
    result: Result,
};

const State = struct {
    values: BoundedMap([64]u8, [32]u8),
    clients: BoundedMap(u128, ClientRecord),
    applied_slot: paxos.Slot,
};
```

The application transition must advance through duplicate log slots even when it suppresses a duplicate mutation:

```
// Design sketch, not repository code.
fn apply(state: *State, slot: paxos.Slot, command: Command) !Result {
    std.debug.assert(slot == state.applied_slot + 1);

    if (state.clients.get(command.client_id)) |record| {
        if (command.request_id <= record.request_id) {
            state.applied_slot = slot;
            return record.result;
        }
    }

    const result = switch (command.operation) {
        .noop => Result.noop,
        .put => try state.values.put(command.key, command.value_hash),
        .remove => state.values.remove(command.key),
        .read_barrier => Result.barrier,
    };
    try state.clients.put(command.client_id, .{
        .request_id = command.request_id,
        .result = result,
    });
    state.applied_slot = slot;
    return result;
}
```

Persist `values`, `clients`, and `applied_slot` atomically and include all three in snapshots. The one-record scheme is correct only with monotonically increasing IDs and at most one outstanding request per client. Concurrent or out-of-order requests need a bounded per-request result table and an explicit garbage-collection rule.

9.2.2 Reads are an application protocol

`committedAt` and `readDecided` inspect local protocol state; they do not prove that a node is still leader or caught up at the instant of a read. A simple linearizable design proposes a `read_barrier` through the leader, waits until its slot is applied locally, then reads the database. This costs consensus. A lease or read-index optimization requires reasoning outside the current API. A follower may serve stale reads only when the service contract permits them and should return a version such as configuration ID and applied slot.

Exercise 17.1. Complete the crash-recovery design: order snapshot verification, journal replay, `Node.restore`, application replay, transport activation, and opening the client listener. Mark the point before which a response would be unsafe.

9.3 Large Example: Regional Control Plane

Status: architecture review exercise, not runnable code.

Assume five voters across three failure zones and many independent routing partitions. One leader and one journal can become a bottleneck, so partition commands by a stable routing key and run one sealed log per shard:

```
const Shard = paxos.ReplicatedLog(Command, .{
    .max_members = 5,
    .window_slots = 32_768,
    .max_batch = 64,
    .max_metadata_bytes = 96,
});
```

1. **Protocol independence**: Each shard has its own ballot, slots, stop sign, and bounds.
2. **Resource isolation**: This exists only if the host also bounds per-shard queues and schedules journal, CPU, and network work fairly.
3. **Parallelism**: Different shard journals may progress concurrently when the storage system supports it.

A command containing only a blob hash creates another ordering obligation: replicate and verify the immutable blob before proposing the command that makes the hash visible, or define deterministic missing-blob recovery before apply. Paxos orders the hash; it does not store the blob.

9.3.1 Regional partition drill

If one zone containing two voters is isolated, the remaining three can form a default majority and campaign. The isolated two cannot make progress and must not accept service writes. They may serve versioned stale reads only if that is an explicit control-plane policy.

After healing, `reconnected` and `requestCatchUp` can repair missing commits; history below a peer's memory floor is served by that peer's host from its journal through `serve_range` requests. If the returning node fell below the cluster's chosen trim anchor, the host must install a verified state image at an anchor and replay the retained suffix before the node votes.

Teach it back. Explain the regional design to an operator using three columns: guaranteed by `ReplicatedLog`, required from the host, and merely a proposed service policy. Place write ordering, blob availability, stale reads, authentication, and snapshot transfer in the correct column.

PART VI

Evidence

We separate proof obligations, repository tests, local measurements, and deployment evidence. Each answers a different question.

10 Validation, Testing, and Operations

Learning contract By the end of this part you should be able to describe what the current test suite establishes, design the missing fault harness, interpret the benchmark without causal overclaiming, and plan recovery drills with observable exit criteria.

10.1 Four kinds of confidence

Evidence	Question answered	What it cannot answer
Invariant argument	Why every allowed transition preserves agreement.	Whether the Zig code and host exactly implement the argument.
Deterministic tests	Whether selected executions and edge cases behave as asserted and remain reproducible.	Whether untested schedules are safe.
Model checking or refinement	Whether all states within a finite model satisfy a specification.	Whether codecs, disks, and production code refine that model unless the relation is established.
Fault drills and telemetry	Whether the deployed host recovers under realistic failures and within its objectives.	A general proof of safety.

Lamport's invariant discipline tells us what to check. Knuth's literate discipline tells us to record why each case exists. A regression test should name the failure schedule and the invariant it protects.

10.2 What the repository tests today

`zig build test` currently covers:

1. ballot ordering, membership rejection, and flexible quorum validation;
2. leader election and consecutive stable-leader proposals;
3. duplicate prepare and accept messages;
4. reordered phase-one entry/completion messages and highest-vote recovery;
5. window backpressure, one-node quorums, batches, ticks, and heartbeats;
6. replay rejection of conflicting values and commits;
7. contiguous learner delivery and chunk-bounded catch-up;
8. tagged-cell reuse, memory floors, trim anchors, and lifetime-journal folds;
9. stop-sign sealing, reconfiguration metadata, and restore behavior.

These are deterministic hand-written schedules. `zig build test` additionally runs the seeded simulator in `sim/simulation.zig` (below) across three-node majority, five-node flexible-quorum, and prioritized configurations, plus two contract gates:

1. `test-misuse` builds three fixture programs in Debug, ReleaseSafe, ReleaseFast, and ReleaseSmall and runs them as child processes. The two misuse fixtures read `messagesSlice` or call `reset` without confirming durability; the gate requires the process to abort with the stable diagnostic on stderr in every mode. The third fixture runs the correct sequence and must exit cleanly, proving the check never misfires.
2. `test-compile-errors` compiles one fixture per invalid comptime option (zero bounds, wire-type overflows, zero tick intervals, batch and metadata limits, pointer-bearing values, derived-capacity overflow on a 32-bit target) and passes only when the compiler rejects it with the expected message.

The repository still lacks end-to-end tests against a real crash-safe journal, corrupted/truncated records, codec version skew, authenticated transport, state-image transfer, client retry recovery, and multi-configuration process restart.

> **A useful critique** The core protocol tests often update in-memory `Node` state and a separate `DurableState`, but they cannot demonstrate that a future production journal has correct framing, checksums, sync semantics, truncation recovery, or atomic application-state updates. Those are required host tests.

10.3 The deterministic fault harness

`sim/simulation.zig` implements the simulator this section previously only sketched. Each seeded run owns the nodes, one append-only write journal per node standing in for the disk, a message bag delivered in random order, and a link-cut matrix. One seeded action chooses among delivering (or dropping or duplicating) an envelope, ticking a node, proposing a uniquely identified command at the leader, cutting or healing a link, reconnecting, and crashing a node at one of three host commit points: before any write is persisted, after a durable **prefix** of the writes with no message sent, or after all writes with only a prefix of the messages sent. Restart replays the journal through `DurableState.apply` and any replay error is itself a failure. The harness also plays the window host: it advances each node's memory floor with a deliberate lag behind the decided prefix, so tagged-cell reuse is exercised, and it answers `serve_range` requests from the per-node journal so catch-up below a memory floor is covered too.

After every observed transition the harness checks: all non-null committed values for a slot agree with a golden first-commit table, committed values were proposed or are the no-op, `promised` never regresses within an incarnation, and the decided prefix never shrinks. Every run ends with a fault-free quiescence phase that must converge on one leader and one decided prefix, which catches liveness regressions, not only safety ones.

Failures print the seed, the step, and a trailing action trace; `zig build sim -- --seed=N --steps=M --verbose` replays a run exactly, and the runner halves the step budget to report a minimal reproduction. CI runs 64 seeds per configuration inside `zig build test`; a nightly pipeline runs `--seeds=10000 --steps=4096`.

This harness found three real defects on its first sweeps: `recordCommit` treated a commit that disagreed with a stale local vote as corruption (legal Paxos whenever the choosing quorum excluded that node), a restarted node whose replayed log was committed-but-undelivered could never re-release its prefix, and phase one did not return decrees a node had learned without voting, so a leader behind on the log could stall forever. Each fix landed with the seed that exposed it.

`specs/Paxos.tla` now complements the harness: a TLC-checked model of the durable state and messages, with the promise-carries-learned-decrees rule included, checking agreement, commit uniqueness, promise dominance, and validity on a finite configuration. The spec's action-to-code mapping is in the conformance appendix and `specs/README.md`; a mechanical refinement relation between the Zig code and the spec is still not established.

> **Exercise 19.1.** Design the smallest schedule that crashes an acceptor after it emits a reply but before a hypothetical host sync. Which oracle might still pass for a while? Which later campaign exposes the broken promise? Turn the schedule into a regression requirement for the host journal test.

10.4 The local CPU benchmark

`zig build benchmark-zig` runs a workload matrix through in-memory nodes: commit modes (one value synchronously, pipelined windows of 8 and 64, batches of 16 and 256), payload sizes (8 B, 64 B, 1 KiB), cluster sizes (3 and 5), and a run with twice the needed consensus-window cells so the cost of exactly-

sized arrays is measured instead of assumed. A timing sample aggregates 16–64 independently initialized stable-leader iterations, making even the fastest sample last well beyond the clock's sub-millisecond noise floor. Seven samples produce the median and min/max span; a separate instrumented pass reports window-average time per value, not request latency. The Zig and OmniPaxos fixtures check every replica; Zig additionally checks its separate durable-state mirrors. The LibPaxos3 fixture validates every accept response and the learner. All three validate the checksum and logical envelope count. `zig build benchmark` adds the pinned OmniPaxos 0.2.2 and LibPaxos3 workloads and the durable benchmark below.

A separate moving-window family (`u64-3n-moving`) exercises what the fixed matrix cannot: it drives 262,144 values through a 1,024-cell window on one global slot line, 256 complete window wraps with memory floors advancing as the host consumes, and records batch-level latency percentiles into the same results protocol. Its check is periodicity, not raw speed: window reuse must cost the same at wrap 256 as at wrap 1, with no wrap-correlated spikes.

`sh benchmarks/run-all.sh` runs everything and writes a machine-readable file under `benchmarks/results/`; the dashboard below is rendered from `latest.json` at book build time. Its environment stamp marks a modified working tree, so a development measurement cannot masquerade as an archival result from the named revision.

Recorded 2026-08-27T08:41:08Z · paxos-zig · AMD Ryzen 9 5950X 16-Core Processor · Linux 7.0.0-28-generic · revision `151eb76` · Zig 0.16.0 · rustc 1.98.0

TIMED
Stable leader through in-process decision

CHECKED
Replica decisions, checksums, envelope counts, journal replay

EXCLUDED
Network, codec, contention, application, client queueing

The comparison at a glance

8-byte values · 3 voters · in-memory · median of 7; lower time is better

Core	Submission shape	env / value	ns / value	sample span
Zig	one at a time	6	105.4	2.7%
OmniPaxos	one at a time	6	933.4	6.1%
LibPaxos3	one + phase-one preexecution	12	588.5	13.5%
Zig	window of 64	6	105.3	0.2%
OmniPaxos	window of 64; coalesced	0.09	70.7	0.9%
Zig	`proposeBatch` of 256	6	107.1	0.8%

Sensitivity matrix

Each cell is median ns/value · logical envelopes/value. Rows change one workload dimension.

Workload	Zig sync	Zig win 8	Omni sync	Omni win 8
8 B · 3 voters	105.4 ns · 6 env	107.8 ns · 6 env	933.4 ns · 6 env	180.9 ns · 0.75 env
64 B · 3 voters	203.4 ns · 6 env	210.8 ns · 6 env	1010.5 ns · 6 env	196.3 ns · 0.75 env
1 KiB · 3 voters	2321.6 ns · 6 env	2303.9 ns · 6 env	1158.8 ns · 6 env	380 ns · 0.75 env
8 B · 5 voters	177.3 ns · 12 env	183.4 ns · 12 env	2506.1 ns · 12 env	410.6 ns · 1.5 env

What the Zig API amortizes

All rows still emit one-value protocol envelopes; batching combines host/API work, not wire values.

Mode	Values / drain	ns / value	env / value
individual	1	105.4	6
pipeline	8	107.8	6
pipeline	64	105.3	6
`proposeBatch`	16	108.7	6
`proposeBatch`	256	107.1	6

The window keeps moving

262,144 values on one global slot line · 1,024-cell window · 256 wraps · floors advancing

Measure	p50	p90	p99	max
batch-average ns / value	100	102	106	197

Overall 101 ns/value across 256 window wraps. Reusing a consensus cell costs the same at wrap 256 as at wrap 1: there is no per-epoch cliff because there is no epoch.

Durability changes the scale

512 values · 3 file journals · writes replay-verified after timing · median of 3

Host policy	fsync / value	µs / value	vs in-memory Zig sync
sync every write-bearing transition	6	9795.3	92926×
group commit, window 8	0.75	1258	11934×

"env" counts logical in-process envelopes, not bytes. OmniPaxos may carry many values in one envelope; Zig and this LibPaxos3 fixture carry one. "sample span" is (maximum − minimum) / median across the seven totals; lower is steadier. The doubled-log-slack control is 107.2 ns/value; compare it with Zig sync above before attributing small differences.

Three findings from the recorded results deserve emphasis. First, longer samples make the min/max span visible and usually small, but elapsed time is still machine state, while envelope counts and checksums are deterministic. Second, the harness shape changes the comparison dramatically. One value at a time measures per-append core overhead. With a window of 64, OmniPaxos coalesces entries into about 0.09 logical envelopes per value and narrows the gap substantially; this Zig core continues to emit six one-value envelopes. `proposeBatch` amortizes API/effect handling but is not a wire-batching primitive, which is why its envelope count does not fall. These are design and harness effects, not evidence of language superiority. Third, LibPaxos3 runs a heavier measured twelve-envelope path that advances phase-one preexecution, so it is labeled rather than equated with the stable-leader paths.

`zig build benchmark-durable` measures the safety contract itself: each node serializes its writes to an append-only file and syncs before any dependent message moves. Only write-bearing transitions sync, setup fsyncs are excluded from the timed count, and every journal is reopened and replayed after timing. The ungrouped path performs six fsyncs per value in this three-node message schedule; grouping windows of eight reduces that to 0.75 and improves elapsed time by roughly four to five times. Read the in-memory numbers with the much larger durable scale in mind. Because this host copies writes and messages into its own batches and runs one shared barrier per group, it declares its protocol through

`paxos.host_managed.Protocol` and carries the effect-order obligation itself; the post-run journal replay is the crash-recovery evidence that obligation requires.

The in-memory benchmarks still exclude real serialization, network delay, contention, snapshots, retries, client admission, and application work. Their numbers must never be presented as service latency or throughput, and none of these workloads isolates language, allocator, algorithm, or implementation quality.

10.5 Capability map: exact boundaries

Capability	Core	Boundary
Multi-Paxos log	Yes	Stable leader skips repeated phase one.
Logical elections	Yes	`tick`; no clock reads or randomized deadlines.
Priority, heartbeat, resend	Yes	Bounded deterministic mechanisms.
Flexible quorum sizes	Yes	Validated by `Membership.init`.
Batch proposal	Yes	Bounded caller input and effect storage.
Log inspection	Yes	Not an application linearizable-read protocol.
Chunk-bounded catch-up	Yes	`learn` and `commit` messages; history below the memory floor is served by the host via `serve_range`.
Global slots and window reuse	Yes	The host licenses cell reuse with `advanceMemoryFloor`; accepted-only cells are never evicted.
Trim anchors	Yes	`installChosenTrim` adopts a chosen trim; deciding and encoding trim records is host work.
Reconfiguration boundary	`ReplicatedLog`	Decides a stop sign; the next configuration continues the same slot line via `initFromStop`.
Journal segments, state images, compaction	No	The core never deletes history; the host owns journals, anchors, and state transfer.
Storage, codec, transport, auth	No	Required from the host.
Client sessions	No	Required for retry semantics.

10.6 Operating drills with exit criteria

Drill	Observe before declaring recovery
Follower crash	Verified journal replay; restored configuration ID; monotonic decided/applied prefix; caught-up peer; no duplicate effect.
Leader crash during vote	New higher ballot; recovery of the greatest accepted value; no conflicting commit; client ambiguity resolved by ID.
Minority partition	No minority writes acknowledged; majority progress if its required quorums remain; returning nodes repair before voting.
Disk full or sync failure	Node stops without publishing a durable claim or client success from the failed batch. A classified request may already have asked a peer to persist; recovery remains safe from verified records.

Drill	Observe before declaring recovery
Corrupt state image	Hash/version rejection; no `beginRecovery` at its anchor; recovery from another verified source.

Track ballot changes, role, current leader hint, decided and applied prefixes, the memory floor and trim anchor, journal sync latency, queue depth, retransmissions, catch-up distance, configuration ID, and client retry counts. `currentLeader()` is a hint for routing and metrics, not a lease certificate.

Teach it back. Pick one production claim such as "survives a leader crash." State the invariant argument, repository test, missing integration test, telemetry, and drill exit criterion needed to support that sentence honestly.

PART VII

Desk reference

Messages, effects, APIs, errors, invariants, exercise answers, and research sources collected beside one another.

11 Consensus Desk Reference

11.1 Message reference

Message	Fields	Meaning
`prepare`	`ballot, first`	Ask an acceptor to promise this ballot and answer the recovery chunk starting at `first`.
`promise`	`ballot, slot, accepted`	Report one accepted slot for phase-one recovery.
`promise_range`	`ballot, anchor, chosen_through, first, last, accepted_count, more`	Describe the answered chunk: how many `promise` entries complete it, the acceptor's chosen prefix and trim anchor, and whether it knows state above `last`.
`accept`	`ballot, slot, value`	Ask an acceptor to vote.
`accepted`	`ballot, slot, decided_through`	Acknowledge a durable vote and report the local released prefix.
`commit`	`slot, value`	Teach a value that a correct sender knows was chosen.
`learn`	`from_slot, count`	Request known commits in a chunk-bounded range from a nonzero slot.
`nack`	`rejected, promised, decided_through`	Reject a stale ballot and report the higher promise and local prefix.
`heartbeat`	`ballot, decided_through`	Leader traffic for a matching durable promise.

11.2 Effect ordering

Write	Fields	Must be durable before
`promise`	`ballot`	`promise`, `promise_range`, or other evidence that depends on the promise.
`accept`	`ballot, slot, value`	`accepted` and any later claim based on that vote.
`commit`	`slot, value`	Commit broadcast from the leader, application delivery, and catch-up claims based on that record.
`trim_anchor`	`trim_id, chosen_trim_slot, history_hash`	Any phase-one answer that vouches the released prefix from the adopted anchor.

Consume one batch as: append writes in order; sync the batch; call `confirmWritesDurable`; send messages; apply released entries in order. Do not call another transition before draining the batch. On append or sync failure, discard the already-mutated live node and restore from verified durable state. Every optimize mode enforces the confirmation step: reading `messagesSlice` first stops the process with `paxos: messagesSlice before confirmWritesDurable`, and resetting an unconfirmed batch stops it with `paxos: reset discarded unconfirmed writes`.

11.3 Core API

Symbol	Contract
`Protocol(Value, Options)`	Creates fixed-membership bounded Paxos types; `Value` must contain no pointer, slice, or reference recursively.
`Membership.init(ids)`	Validates nonzero unique IDs and quorum sizes.
`Node.init`, `initWithPriority`	Bootstrap an empty member.
`Node.restore`, `restoreWithPriority`, `restoreAt`	Restore protocol durable state, optionally resuming floor and delivery at the host's consumed prefix; application state remains host-owned.
`Node.continueAt`	Start an empty node at a floor on the same global slot line with an inherited trim anchor.
`campaign(noop)`, `tick(noop)`	Start phase one explicitly or advance deterministic liveness counters.
`step(envelope)`	Process one authenticated, decoded member envelope for this recipient.
`propose`, `proposeBatch`	Assign slots after phase one; return `NotLeader` otherwise.
`reconnected`, `requestCatchUp`	Repair one peer path or emit a chunk-bounded `learn` request.
`committedAt`, `readDecided`	Inspect the local decided log; not an application read protocol.
`currentLeader`, `decidedThrough`	Diagnostic leader hint and contiguous released prefix.
`advanceMemoryFloor`, `memoryFloor`	Record that the host durably consumed the released prefix, licensing tagged-cell reuse below it.
`installChosenTrim`, `trimAnchor`, `beginRecovery`	Adopt a chosen trim anchor, read it back, or reset onto an installed state image at an anchor.
`Effects.init/reset`	Initialize or clear active counts without clearing backing storage. Public transitions reset automatically. `reset` stops the process if the batch holds unconfirmed writes.
`Effects.confirmWritesDurable`	Host statement that the pending batch is durable; required before `messagesSlice` in every optimize mode.
`DurableState.apply`	Strict replay semantics for one configuration's ordered `Write` records; not a disk format.
`DurableState.replayFold`	Fold a lifetime journal across configuration changes and window reuse: promises fold to their maximum, stale accepts in reused cells are skipped, commits and trim anchors stay strict.
`host_managed.Protocol(Value, Options)`	Same types without the runtime ordering check; the host owns the durability boundary. An audited exception for grouped-barrier hosts only.

11.4 Replicated-log API

`ReplicatedLog(Value, Options)` wraps `Protocol` with an `Entry` union of `command` and `stop`. It forwards `window_slots` and `recovery_chunk_slots` to the core and adds `max_batch` and `max_metadata_bytes`.

Symbol	Contract
`init(id, configuration_id, membership)`	Start a nonzero configuration.
`campaign`, `tick`, `step`	Core operations with application `Value` no-ops wrapped as commands.
`append`, `appendBatch`	Propose commands unless a stop is pending or decided.
`reconfigure(id, members, metadata)`	Propose a strictly newer stop sign and seal local appends; the only sealing path.
`isReconfigured()`	Return the **decided** stop, which permits host handover. A replayed or served stop naming a configuration the node already runs is completed history and never re-seals the log.
`initFromStop(id, stop, stop_slot, anchor, membership, priority)`	Validate the stop's member slice and start its configuration at the stop slot on the same global slot line, carrying the inherited trim anchor.
`continueAt`, `restoreAt`	Continue or restore a configuration at a floor on the same slot line.
`advanceMemoryFloor`, `installChosenTrim`, `trimAnchor`, `beginRecovery`	Window and trim pass-throughs to the core node.
`read`, `decidedThrough`, `currentLeader`	Inspect this configuration's resident window state.
`configurationId`	Return the host-supplied durable configuration identity.

11.5 State lifetime

Field	Meaning	Lifetime
`durable.promised`	Highest ballot this acceptor permits.	Durable
`durable.anchor`	Adopted chosen-trim anchor.	Durable
`durable.cells[].accepted`	Highest stored ballot/value per resident slot; cells are slot-tagged.	Durable
`durable.cells[].committed`	Known chosen value per resident slot.	Durable
`role`, `ballot`, `leader_hint`	Local leadership state.	Volatile
`highest_observed_round`	Largest round seen in higher traffic.	Volatile
`next_slot`	Next proposal slot.	Rebuilt
`delivered_through`	Prefix released in this process.	Volatile
`memory_floor`	Host-licensed cell-reuse floor.	Rebuilt via `restoreAt`
`election`, `promise_seen`, `recovered`	Phase-one chunk and recovery state.	Volatile
`lead`	Leader proposal and acknowledgement state per live slot.	Volatile

Field	Meaning	Lifetime
`configuration_id`	Replicated-log configuration identity.	Host must persist
`stop_pending`, `stop_sign`	Seal state derived from core state/ effects.	Rebuilt

11.6 Error reference

Error	Meaning
`EmptyMembership`, `TooManyMembers`	Membership length violates bounds.
`InvalidNodeId`, `DuplicateNodeId`	ID zero or duplicate ID.
`InvalidReadQuorum`, `InvalidWriteQuorum`	Configured size is zero or exceeds actual membership.
`NonIntersectingQuorums`	`read + write <= member_count`.
`NotMember`, `WrongRecipient`, `InvalidPeer`	Invalid envelope or peer identity for this operation.
`NotVoter`, `NotLearner`, `LearnerIsVoter`, `LearnerMessageForbidden`	A role-restricted operation or message reached the wrong kind of node.
`ConfigurationMismatch`	The message's configuration ID differs from the local one.
`NotLeader`	Proposal attempted before completed phase one.
`CampaignDisabled`	This voter is configured to never start elections.
`BallotExhausted`	No greater `u64` round exists.

Error	Meaning
`InvalidSlot`	Slot zero, or a floor or trim claim above the delivered prefix.
`WindowFull`	Every consensus cell holds a live slot. Transient backpressure: retry after the memory floor advances.
`Trimmed`	The requested history is below the memory floor; recover it from the host journal, never from a reused cell.
`GlobalSlotExhausted`	The 64-bit global slot space is exhausted; it never wraps. Terminal.
`EmptyBatch`, `SlotBufferTooSmall`, `BatchTooLarge`	Invalid batch input or output capacity.
`ReadBufferTooSmall`	Caller output cannot hold the available prefix.
`InvalidPromise`, `MissingNoop`, `MissingProposedValue`	Incomplete or inconsistent leader recovery state.
`PromiseRegression`, `ConflictingValue`, `ConflictingCommit`, `ConflictingChosenValue`	Replay or transition contradicts durable monotonicity or

Error	Meaning
	value uniqueness.
`TrimRegression`	A trim anchor moved backward or conflicts with the adopted one.
`WindowOverrun`	A journal record addresses a cell still occupied by an earlier slot: the journal ran past the window without an anchor.
`InvalidConfigurationId`, `ConfigurationIdRegression`	Zero or non-newer configuration ID.
`ConfigurationIdExhausted`	No next `u64` configuration ID.
`MetadataTooLarge`, `LogSealed`	Stop metadata exceeds its bound or the current configuration no longer accepts commands.

`paxos.explainError(err)` supplies operator-oriented prose. Errors from host I/O have no protocol-specific explanation and must retain their original storage or transport context.

11.7 Formula sheet

Quantity	Formula	Example
Majority	`floor(N / 2) + 1`	`N=5` gives 3.
Majority crash tolerance	`N - majority`	`5 - 3` gives 2.
Uniform flexible safety	`Q1 + Q2 > N`	`4 + 2 > 5`.
Stable-path logical messages	`3(N - 1)`	`N=3` gives 6: accepts, accepted replies, commits.

Quantity	Formula	Example
Window backpressure bound	`next_slot - memory_floor <= window_slots`	Beyond it, `propose` returns `WindowFull`.
Accept payload egress	`payload_bytes * peers * proposals_per_second`	Excludes headers, retransmits, and commit payloads.

11.8 Invariants for review

1. Complete ballots are unique and totally ordered.
2. Every allowed phase-one quorum intersects every allowed phase-two quorum.
3. An acceptor never accepts below its durable promise.
4. One ballot and slot never carry two different values.
5. A new leader selects the value with the greatest accepted ballot in each slot across a complete phase-one quorum, or uses a no-op for a required hole.
6. A value is chosen when a phase-two quorum has accepted it; knowledge and commit dissemination may occur later.
7. A committed slot never changes value.
8. Application release is a contiguous slot prefix.
9. Every effect-dependent message waits for all writes in its batch to sync.
10. A global slot number is never reused. A physical cell is retagged only for a chosen slot the host has released below the memory floor, and an elected leader never proposes at or below the greatest quorum-reported trim anchor or chosen prefix.

11.9 Answers to selected exercises

11.9.1 Exercise 1.1

A majority of five is three. Two nodes may be unavailable while three remain.

11.9.2 Exercise 2.1

`{A1, A2}` and `{A3, A4}` do not intersect. They could choose conflicting values independently if both were legal quorums.

11.9.3 Exercise 4.1

Choose `apple` from ballot 9. Do not count values. Knowledge that ballot 3 was chosen does not alter the mechanical rule; safety implies the ballot-9 vote is also `apple` if the history is legal.

11.9.4 Exercise 4.2

The shared acceptor may have any name; both value blanks are `x`.

11.9.5 Exercise 8.1

`tea` is already chosen when the quorum accepts, before the commit record or broadcast. Durable accepted records in every future intersecting recovery quorum force later leaders to preserve `tea`; `coffee` cannot be chosen.

11.9.6 Exercise 11.1

Twenty-two: slots 83 through 104. At slot 105, `next_slot - memory_floor` would exceed the 64-cell window and `propose` returns `WindowFull`. The host frees cells by durably consuming released entries and calling

`advanceMemoryFloor`. The cells for 81 and 82 hold accepted but unchosen votes; an accepted-only cell is never evicted, because discarding the vote could let a later leader choose a different value for those slots.

11.9.7 Exercise 14.1

With `N=7` and `Q2=3`, phase one must satisfy `Q1+3>7`, so `Q1=5` is the minimum. Leader replacement can tolerate two unavailable members.

11.10 Glossary

Term	Meaning
Accepted	One acceptor durably voted for a ballot, slot, and value.
Chosen	A phase-two quorum accepted one value in one slot.
Committed	A learner durably recorded a value known to be chosen.
Applied	The host state machine executed a committed entry.
Ballot	A unique ordered proposal attempt `(round, priority, node)`.
Configuration	One fixed voter set, sealed only by a decided stop sign; the next configuration continues the same slot line.
Leader	A candidate that completed phase one for its ballot.
Memory floor	The greatest slot whose released entry the host has durably consumed, licensing cell reuse below it.
No-op	A host-defined value that consumes a slot without application work.
Promise	A durable refusal to accept lower ballots.
Quorum	A voter subset participating in a phase.
Slot	A one-based `u64` position on the global log; never reset or reused.
Stop sign	A decided entry that names the next configuration and seals the old one.
Trim anchor	A chosen record stating that every slot at or below it is chosen under a bound history hash.
Window	The fixed array of slot-tagged cells holding resident consensus state; it bounds residency, not history.

11.11 Research and design sources

11.11.1 Paxos, state machines, and proof structure

1. Leslie Lamport, "The Part-Time Parliament" develops the replicated state-machine setting and the ballot invariants.
2. "Paxos Made Simple" derives the proposal rules from consensus safety requirements.
3. "The Paxos Algorithm" teaches three levels: goal, high-level state algorithm, and message algorithm.
4. "Teaching Concurrency" argues that state, next-state relations, and invariants are the foundation for understanding concurrent systems.
5. "How to Write a Proof" motivates hierarchical structure for long reasoning.
6. "Reconfiguring a State Machine" describes stop signs, padding, and configuration handover.
7. "Stoppable Paxos" supplies a more formal stop construction.
8. Howard, Malkhi, and Spiegelman, "Flexible Paxos" states the cross-phase quorum intersection trade-off.

11.11.2 Learning design

1. Sweller, van Merrienboer, and Paas, "Cognitive Architecture and Instructional Design" reviews limited working memory, schemas, worked examples, split attention, and redundancy.
2. Ward and Sweller, "Structuring Effective Worked Examples" studies attention and cognitive load in worked-example design.
3. Renkl, Atkinson, and Große, "How Fading Worked Solution Steps Works" motivates transition from examples to completion and independent problems.
4. Chi et al., "Self-Explanations" supplies the empirical basis for explaining why each worked step follows.
5. Roediger and Karpicke, "Test-Enhanced Learning" motivates retrieval after a delay rather than relying on rereading familiarity.

11.11.3 Feynman and Knuth as design influences

1. Richard Feynman, "What Is Science?" models concrete explanation, observation, doubt, and admitting when an explanation fails. It is a lecture, not an instructional trial.
2. Caltech, "The Power of Teaching" presents a modern Feynman-inspired teach-to-learn routine and explicitly links vagueness to a study gap.
3. Donald Knuth, "Literate Programming" treats programs as literature addressed to people. This book adopts its human-first exposition without claiming an empirical learning effect or generating Zig from prose.

11.12 Closing note

Paxos is not a bag of messages. It is one preservation rule carried through time. Quorum intersection finds a witness. Stable storage lets the witness remember. Phase one asks. Phase two records. Slots order decisions. The host makes those abstract facts physical with sync, codecs, identity, deterministic application, snapshots, and recovery.

Teach it back. Close the book and explain the entire protocol in six sentences. Reopen this invariant list, mark the first omitted fact, and revise only that sentence. The gap—not the first draft—is the learning result.

PART VIII

Conformance

One table from the paper to the code to the test oracles to the model: every safety-relevant rule is traceable in all four places.

12 Lamport Conformance Appendix

This appendix maps the basic protocol of *The Part-Time Parliament* (section 2.3 and the appendix's algorithm) and the multi-decree refinements (section 3) to `src/protocol.zig`, to the simulator's runtime oracles in `sim/simulation.zig`, and to the TLC-checked model in `specs/Paxos.tla`. Line references drift; function names are the stable anchors.

12.1 The basic protocol, step by step

Paper	Rule	Code (`protocol.zig`)	Spec action
Step 1	Choose a ballot greater than `lastTried`, owned by this priest.	`startCampaign`: `round = max(own, promised, observed) + 1` with the node ID as tie-breaker; `Ballot.order`	`Prepare`
Step 2	On `NextBallot(b)` with `b >= nextBal`, set `nextBal` and reply `LastVote` with the highest vote.	`onPrepare`: `lessThan` guard, `Write.promise`, per-slot `promise` replies, `promise_range` chunk descriptor; lower ballots are nacked	`Promise`
Step 3	With `LastVote` from a majority, propose the decree of the highest-ballot vote, else any decree (B3).	`onPromise` keeps the highest-ballot vote per slot; `maybeResolveChunk` requires complete chunk descriptions from a read quorum; `resolveChunk` re-drives recovered slots above the quorum's trim and chosen fences and fills gaps with the no-op decree	`Accept` and `ChoosableFor`
Step 4	On `BeginBallot(b, d)` with `b >= nextBal`, cast the vote and record it in the ledger.	`onAccept`: `lessThan` guard, `Write.accept` persisted before the `accepted` reply; same-ballot conflicts are `ConflictingValue`	`Vote`
Step 5	With `Voted` from every quorum member, the decree passes.	`onAccepted`: distinct-member count against `writeQuorum()`, then `recordCommit`	`Decide`
Step 6	On `Success(d)`, write the decree in the ledger.	`onCommit` / `recordCommit`; `emitContiguous` releases the decided prefix in order	`Learn`

12.2 Multi-decree refinements

Paper	Rule	Code
Section 3.1	One `NextBallot(b, n)` covers every decree instance; the reply carries votes for all instances after `n`.	`onPrepare` answers the bounded chunk `[first, first + chunk - 1]`; `promise_range` carries the count so leadership waits for a complete chunk per member, and its `more` flag drives the next `prepare`
Section 3.1	The reply also reports already-passed decrees the president may be missing.	`onPrepare` reports a learned-only decree as a zero-ballot vote, which loses to every real vote and can never override the choosing quorum
Section 3.1	Fill gaps with the harmless "olive-day" decree.	`resolveChunk` proposes the caller's no-op for unrecovered slots below known state; slots at or below a quorum-reported trim anchor or chosen prefix are released history, never gaps
Section 2.2 (B1)	Ballot numbers are partitioned among priests.	`Ballot = (round, priority, node)`; the node component makes reuse across owners impossible
Section 2.2 (B2)	Any two quorums intersect.	`Membership.init` rejects `read + write <= count` (`NonIntersectingQuorums`); majority by default, flexible quorums allowed
Progress	Decrees eventually reach every ledger in the Chamber.	Leader heartbeats advertise `decided_through`; a behind follower replies `learn`, which also corrects the leader's stale view of that follower; the leader re-releases its own prefix on election

12.3 Durable state

The paper's ledger variables map onto `DurableState`. The host must sync them before releasing messages. The always-on guard in `Effects` enforces the required call order in every build mode:

Paper	Code	Persisted by
`nextBal`	`durable.promised`	`Write.promise` (and every `Write.accept`, which also advances the promise)
`prevBal`, `prevDec`	`durable.cells`, slot-tagged, read through `acceptedAt(slot)`	`Write.accept` before the `accepted` reply may be sent

Paper	Code	Persisted by
`outcome`	`durable.cells`, slot-tagged, read through `committedAt(slot)`	`Write.commit` before the commit broadcast
`lastTried`	reconstructed	not persisted directly: recovered as `promised.round + 1`, which is safe because any accept sent under a ballot implies the promise was durably advanced first

A commit that disagrees with a stale local vote is legal and accepted (the choosing quorum may not have included this node); two commits for one slot that disagree are corruption (`ConflictingCommit`). The paper never compares `outcome` against `prevDec`, and neither does the code.

The ledger has one extension the paper does not need: `Write.trim_anchor` persists an adopted chosen-trim anchor (ZDS 0011). After its journal bytes below the anchor are gone, an acceptor still answers phase one correctly, because the anchor states that the whole prefix is chosen, and the leader selection rule never proposes into it.

12.4 The oracles that watch the same rules

The simulator checks, after every observed transition: agreement against a golden first-commit table, validity (proposed or no-op), promise monotonicity per incarnation, monotone decided prefixes, journal replay without error, and post-fault convergence. The TLC model checks `Agreement`, `CommitUniqueness`, `PromisedDominatesVotes`, and `Validity` over all reachable states of its finite configuration. A second model, `specs/GlobalTrim.tla`, covers what the paper predates: the slot-tagged window, host-licensed eviction, the trimmed-acceptor election fences, the conservative cluster trim, and the joiner lease lifecycle. The mapping from spec action to handler is one-to-one with the first table above, so a change to any handler should update this appendix, the simulator's oracles, and the spec together.