Build Your Own Raft: Consensus Algorithm Design Document

Overview

This system implements the Raft consensus algorithm, enabling multiple servers to agree on a sequence of log entries despite partial failures and network delays. The key architectural challenge is maintaining safety (correctness) while maximizing availability and understandability for developers learning distributed systems fundamentals.

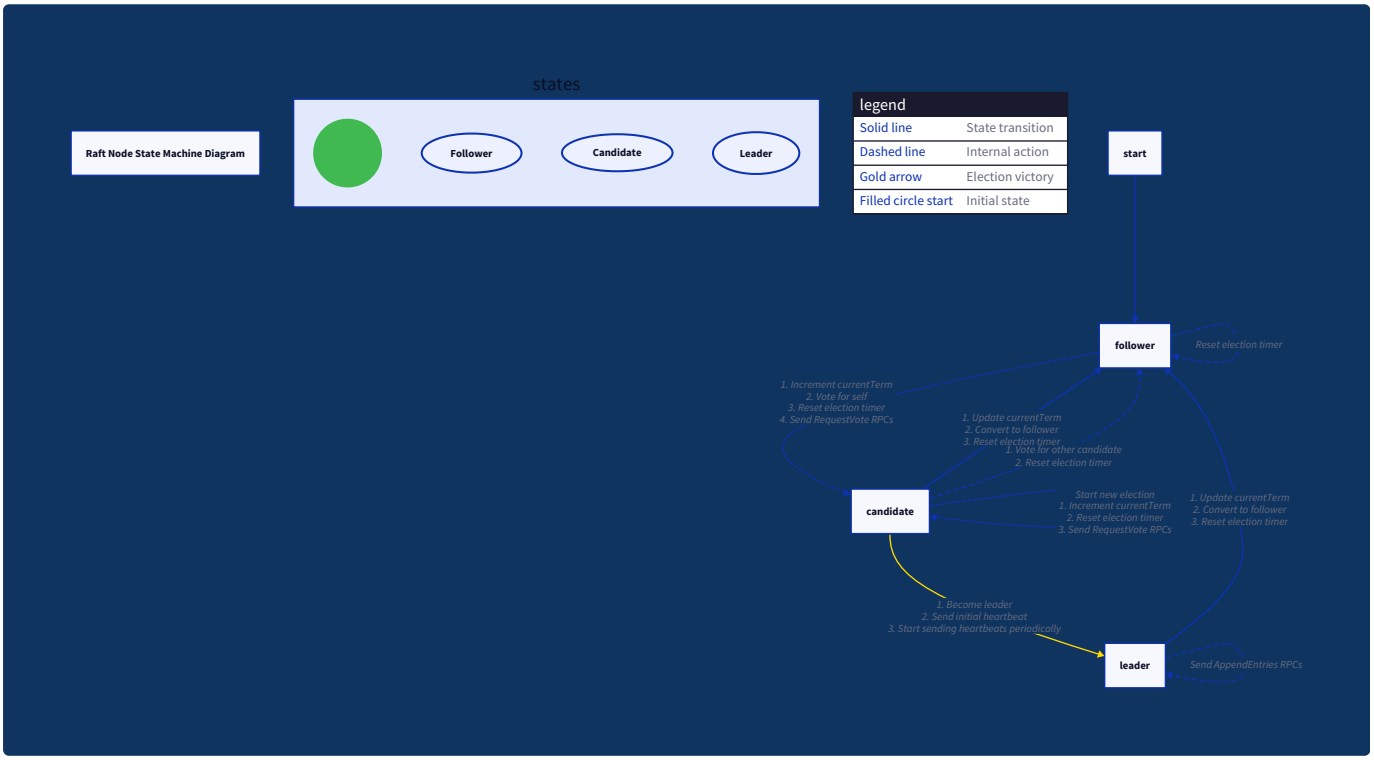
> This guide is meant to help you understand the big picture before diving into each milestone. Refer back to it whenever you need context on how components connect.

1. Context and Problem Statement

> **Milestone(s):** 1 (Leader Election), 2 (Log Replication), 3 (Safety Properties), 4 (Cluster Membership Changes)
>
> This foundational section corresponds to all four milestones, establishing the core problem that the Raft consensus algorithm solves and why it's uniquely suitable for educational implementation.

1.1 The Replicated State Machine Analogy

Imagine a team of accountants responsible for maintaining an identical set of financial ledgers for a large corporation. Each accountant has their own copy of the ledger, and whenever a financial transaction occurs—a payment, receipt, or adjustment—all accountants must record that transaction in exactly the same order in their respective ledgers. If even one accountant records transactions in a different sequence, the ledgers will diverge, leading to financial discrepancies that could compromise the entire organization's integrity.



This scenario captures the essence of **distributed consensus**: multiple independent processes (the accountants) must agree on a sequence of operations (the transactions) despite occasional failures, network delays, or temporary unavailability of some members. The challenge is ensuring that all correct participants apply the same operations in the same order, creating the illusion of a single, fault-tolerant ledger.

> The critical insight in Raft is that this problem becomes tractable by appointing a **single leader** at any given time. Just as a head accountant might dictate the sequence of transactions to the team, the Raft leader sequences client commands and orchestrates their replication to followers. This leader-centric approach dramatically simplifies the protocol's logic and understandability compared to peer-to-peer approaches.

The replicated state machine model formalizes this intuition. Each server (accountant) maintains three key components:

1. **Log**: An append-only sequence of commands (transactions) that have been proposed but not necessarily agreed upon.
2. **State Machine**: The actual business logic (like a key-value store or configuration manager) that applies committed commands from the log.
3. **Consensus Module**: The Raft algorithm itself, responsible for ensuring all servers' logs eventually contain the same commands in the same order.

When a client submits a command (e.g., "set X = 5"), the leader appends it to its log, then replicates it to follower logs. Once a **majority (quorum)** of servers have stored the command in their logs, the leader considers it **committed** and applies it to its state machine. Followers subsequently apply committed commands as they learn about them. This process ensures that even if the leader fails, any new leader elected will have all committed commands in its log, preserving consistency.

1.2 Why Consensus is Difficult: The CAP Perspective

Distributed consensus sits at the heart of the **CAP theorem**, which states that in the presence of a network partition (P), a distributed system can provide either Consistency (C) or Availability (A), but not both simultaneously. Raft is explicitly designed as a **CP system**: it prioritizes consistency over availability during partitions.

Consider our accountant team during a network partition that splits them into two isolated groups. If both groups continue accepting and processing transactions independently, their ledgers will irreconcilably diverge when the network heals—violating consistency. Raft's solution is to allow only the majority side of the partition (if one exists) to make progress by electing a leader and processing commands. The minority side becomes unavailable for writes, preserving the consistency guarantee at the cost of temporary unavailability.

> **Design Principle:** Raft's CP orientation makes it ideal for applications where correctness is paramount, such as coordination services (like etcd or ZooKeeper), primary-backup databases, and configuration management systems. The temporary unavailability during partitions is an acceptable trade-off for these use cases.

The difficulties in achieving consensus extend beyond partitions:

Challenge	Description	Raft's Approach
Partial Failures	Some servers may crash, stop responding, or experience network delays while others continue operating.	Uses **timeouts** and **heartbeats** to detect failures and trigger leader elections.
Asynchronous Networks	Messages can be delayed, reordered, or duplicated arbitrarily.	Relies on **terms** (monotonically increasing numbers) to distinguish old messages from current ones.
Concurrent Leadership	Two servers might believe they are the leader simultaneously (split brain).	Ensures at most one leader per term through majority voting and term comparison rules.
State Reconciliation	After a failure, rejoining servers must catch up without violating consistency.	Uses **log matching properties** and consistency checks during replication to bring lagging servers up to date.

The interplay of these challenges makes building a correct consensus algorithm notoriously difficult. Even small implementation errors can lead to subtle bugs that only manifest under specific failure sequences, making testing and verification essential.

1.3 Pre-Raft Landscape: Paxos and its Challenges

Before Raft's introduction in 2014, **Paxos** (published by Leslie Lamport in 1989) was the predominant consensus algorithm used in both academia and industry. Paxos solves the same consensus problem and is provably correct, but it gained a reputation for being exceptionally difficult to understand and implement correctly. Many engineers quipped that "Paxos is simple, but understanding it isn't."

The fundamental issue with Paxos wasn't its correctness but its **pedagogical approach**. Paxos presents consensus as a single-round voting protocol on individual values, leaving the construction of a multi-decree log (necessary for practical systems) as an exercise for the reader. This

decomposition led to multiple interpretations and complex extensions (Multi-Paxos, Cheap Paxos, Fast Paxos) that further increased the mental burden on implementers.

> **Architecture Decision Record: Choosing Raft Over Paxos for Educational Implementation**
>
> - **Context:** We need to select a consensus algorithm for implementation in an educational context where understandability and incremental implementation are primary goals.
> - **Options Considered:**
>   1. **Implement Classic Paxos:** The original single-decree algorithm with extensions to a log.
>   2. **Implement a Paxos variant** (e.g., Multi-Paxos with leader election).
>   3. **Implement Raft** as described in the "In Search of an Understandable Consensus Algorithm" paper.
> - **Decision:** Implement Raft.
> - **Rationale:**
>   - **Explicit decomposition:** Raft separates leader election, log replication, and safety into distinct, well-defined phases that map directly to implementation milestones.
>   - **Strong leader model:** The single-leader approach simplifies the replication logic and matches the mental model of many practical systems.
>   - **Understandability focus:** The Raft paper was specifically written to be more accessible, with clear explanations and visualizations.
>   - **Pedagogical resources:** Extensive educational materials, visualizations, and reference implementations exist for Raft.
> - **Consequences:**
>   - Learners can implement the algorithm incrementally, validating each milestone before moving to the next.
>   - The resulting implementation will be easier to debug, test, and extend with features like log compaction or membership changes.
>   - Some production systems use Paxos variants, so learners may need additional study to understand those systems.

The following table contrasts the two algorithms across dimensions critical for learning and implementation:

Dimension	Paxos	Raft	Impact on Learning
Decomposition	Single-decree consensus first, then extended to logs (Multi-Paxos)	**Log replication as first-class concept** from the beginning	Raft's structure maps directly to implementation modules (election, replication, safety)
Leadership	Roles: Proposer, Acceptor, Learner. Any node can propose.	**Strong leader**: Only leader can propose entries for a given term	Simplifies replication logic; easier to reason about command flow
State Machine	Not specified in original paper; left as implementation detail	**Explicit state machine** with commit and apply indices	Clear separation between consensus and application logic
Configuration Changes	Complex; requires separate protocol or Paxos instance	**Integrated via log entries** (joint consensus)	Unified approach using the same replication mechanism
Understandability	Notoriously difficult; "The Part-Time Parliament" uses parliamentary analogy	**Designed for understandability** with clear terminology and visualizations	Learners spend less time deciphering the algorithm and more time implementing
Implementation Complexity	High; many subtle corner cases in production implementations	**Moderate**; well-defined states and transitions reduce corner cases	Achievable in educational timeframe with fewer "gotchas"

Raft's design choices—particularly its strong leadership model and log-centric approach—make it exceptionally suitable for educational implementation. Each component of the algorithm corresponds to a clear, testable milestone, allowing learners to build confidence incrementally while developing an intuition for distributed systems fundamentals.

The remainder of this design document will guide you through implementing Raft's four core components, with each section providing the mental models, architectural decisions, and implementation guidance needed to build a working, correct consensus system.

1.4 Implementation Guidance

> **Note:** This section focuses on foundational concepts rather than code implementation, so the Implementation Guidance here is brief. Subsequent sections will provide extensive code examples.

Technology Recommendations:

Component	Simple Option (For Learning)	Advanced Option (For Production)
Transport Layer	HTTP/1.1 with JSON serialization using `net/http`	gRPC with Protocol Buffers for binary efficiency
Serialization	Go's `encoding/json` package	Protocol Buffers or Cap'n Proto
Persistence	Append-only log file with `os.File`	Memory-mapped files or specialized WAL
Concurrency	Goroutines with channels and `sync.Mutex`	Fine-grained locking or actor model
Testing	Go's built-in `testing` package with `net/http/httptest`	Property-based testing with `gopter`

Recommended Project Structure:

While the full architecture will be detailed in Section 3, begin with this minimal structure to organize your implementation from the start:

```
build-your-own-raft/
├── README.md
├── go.mod
├── cmd/
│   └── raft-node/
│       └── main.go                # Entry point for a Raft node
└── internal/
    ├── raft/
    │   ├── raft.go                # Core Raft struct and main loop
    │   ├── state.go               # State definitions (Follower, Candidate, Leader)
    │   ├── log.go                 # Log storage and management
    │   ├── persistence.go         # Persistent state saving/loading
    │   ├── rpc.go                 # RPC structs and handlers
    │   └── raft_test.go           # Unit tests
    ├── transport/
    │   ├── http.go                # HTTP transport implementation
    │   └── interface.go           # Transport interface
    └── statemachine/
        └── interface.go           # State machine interface
```

Infrastructure Starter Code:

Since the transport layer is not the core learning goal, here's a complete, reusable HTTP transport implementation that you can use throughout the project:

GO

```go
// internal/transport/http.go

package transport

import (
    "encoding/json"
    "fmt"
    "net/http"
    "sync"
)

// Message represents a generic RPC message between nodes.
type Message struct {
    Type    string          `json:"type"`    // "RequestVote", "AppendEntries", etc.
    From    int             `json:"from"`    // Sender ID
    To      int             `json:"to"`      // Recipient ID
    Payload json.RawMessage `json:"payload"` // Serialized RPC args
}

// Handler processes incoming messages and returns a response.
type Handler func(msg Message) (interface{}, error)

// HTTPServer provides HTTP-based communication between Raft nodes.
type HTTPServer struct {
    nodeID   int
    addr     string
    handler  Handler
    server   *http.Server
    mu       sync.Mutex
    clients  map[int]*http.Client // Cache of HTTP clients to other nodes
}

// NewHTTPServer creates a new HTTP transport server.
func NewHTTPServer(nodeID int, addr string, handler Handler) *HTTPServer {
    return &HTTPServer{
        nodeID:  nodeID,
        addr:    addr,
        handler: handler,
        clients: make(map[int]*http.Client),
```

```
    }
}

// Start begins listening for incoming RPCs.
func (s *HTTPServer) Start() error {
    mux := http.NewServeMux()
    mux.HandleFunc("/rpc", s.handleRPC)

    s.server = &http.Server{
        Addr:    s.addr,
        Handler: mux,
    }

    go func() {
        if err := s.server.ListenAndServe(); err != nil && err != http.ErrServerClosed {
            fmt.Printf("HTTP server error: %v\n", err)
        }
    }()

    return nil
}

// handleRPC processes incoming HTTP requests.
func (s *HTTPServer) handleRPC(w http.ResponseWriter, r *http.Request) {
    var msg Message
    if err := json.NewDecoder(r.Body).Decode(&msg); err != nil {
        http.Error(w, err.Error(), http.StatusBadRequest)
        return
    }

    response, err := s.handler(msg)
    if err != nil {
        http.Error(w, err.Error(), http.StatusInternalServerError)
        return
    }

    w.Header().Set("Content-Type", "application/json")
```

```go
    json.NewEncoder(w).Encode(response)
}

// Send delivers a message to another node.
func (s *HTTPServer) Send(to int, msg Message) (interface{}, error) {
    s.mu.Lock()
    client, ok := s.clients[to]
    if !ok {
        client = &http.Client{Timeout: 5 * time.Second}
        s.clients[to] = client
    }
    s.mu.Unlock()

    // In practice, you'd need to know the target node's address
    // For now, we assume a simple addressing scheme
    targetAddr := fmt.Sprintf("http://node-%d:8080/rpc", to)

    var buf bytes.Buffer
    if err := json.NewEncoder(&buf).Encode(msg); err != nil {
        return nil, err
    }

    resp, err := client.Post(targetAddr, "application/json", &buf)
    if err != nil {
        return nil, err
    }
    defer resp.Body.Close()

    var result interface{}
    if err := json.NewDecoder(resp.Body).Decode(&result); err != nil {
        return nil, err
    }

    return result, nil
}

// Stop shuts down the HTTP server.
```

```
func (s *HTTPServer) Stop() error {
    if s.server != nil {
        return s.server.Shutdown(context.Background())
    }
    return nil
}
```

Core Logic Skeleton - Raft Node Foundation:

Here's the minimal skeleton for your core Raft node structure. You'll expand this in subsequent milestones:

```go
// internal/raft/raft.go

package raft

import (
    "sync"
    "time"
)

// NodeState represents the three possible states of a Raft node.
type NodeState int

const (
    Follower NodeState = iota
    Candidate
    Leader
)

// RaftNode represents a single node participating in the Raft consensus protocol.
type RaftNode struct {
    mu sync.Mutex

    // Persistent state (must survive crashes)
    currentTerm int        // Latest term this node has seen
    votedFor    int        // Candidate ID that received vote in current term (or -1)
    log         []LogEntry // Log entries containing commands

    // Volatile state (reinitialized after restart)
    commitIndex int // Index of highest log entry known to be committed
    lastApplied int // Index of highest log entry applied to state machine

    // Volatile state on leaders (reinitialized after election)
    nextIndex  []int // For each follower, index of next log entry to send
    matchIndex []int // For each follower, index of highest log entry known to be replicated

    // Other node metadata
    id          int           // Unique identifier for this node
    peers       []int         // IDs of other nodes in the cluster
    state       NodeState     // Current state (Follower, Candidate, Leader)
```

```
    stateMachine StateMachine // Application state machine

    // Election timeout
    electionTimeout       time.Duration
    electionTimeoutTimer  *time.Timer
    lastHeartbeatReceived time.Time

    // Transport for communicating with other nodes
    transport Transport
}

// NewRaftNode creates and initializes a new Raft node.
func NewRaftNode(id int, peers []int, sm StateMachine, transport Transport) *RaftNode {
    node := &RaftNode{
        id:        id,
        peers:     peers,
        state:     Follower,
        votedFor:  -1, // -1 indicates no vote cast
        log:       []LogEntry{{Term: 0, Index: 0}}, // Dummy entry at index 0
        nextIndex: make([]int, len(peers)),
        matchIndex: make([]int, len(peers)),

        stateMachine: sm,
        transport:    transport,

        // Set initial election timeout (150-300ms is typical in tests)
        electionTimeout: time.Duration(150+rand.Intn(150)) * time.Millisecond,
    }

    // Initialize timers
    node.resetElectionTimeout()

    // Start the main loop in a goroutine
    go node.run()

    return node
```

```go
}

// run is the main event loop for the Raft node.
func (node *RaftNode) run() {
    for {
        node.mu.Lock()
        state := node.state
        node.mu.Unlock()

        switch state {
        case Follower:
            node.runFollower()
        case Candidate:
            node.runCandidate()
        case Leader:
            node.runLeader()
        }
    }
}

// runFollower handles the behavior of a node in Follower state.
func (node *RaftNode) runFollower() {
    // TODO Milestone 1: Implement follower logic
    // 1. Wait for either:
    //    a. Election timeout -> transition to Candidate
    //    b. Valid RPC from leader or candidate
    // 2. On election timeout, transition to Candidate state
    // 3. On receiving RPC, process according to RPC type
    // Hint: Use select with channels or a timer
}

// runCandidate handles the behavior of a node in Candidate state.
func (node *RaftNode) runCandidate() {
    // TODO Milestone 1: Implement candidate logic
    // 1. Increment currentTerm
    // 2. Vote for self
    // 3. Reset election timer with new random timeout
```

```
    // 4. Send RequestVote RPCs to all peers
    // 5. Collect votes until:
    //    a. Majority votes received -> become Leader
    //    b. Election timeout -> start new election
    //    c. Discover higher term -> become Follower
    //    d. Valid AppendEntries from current leader -> become Follower
}

// runLeader handles the behavior of a node in Leader state.
func (node *RaftNode) runLeader() {
    // TODO Milestone 2: Implement leader logic
    // 1. Send initial empty AppendEntries (heartbeat) to all followers
    // 2. Start periodic heartbeats (every 50-100ms)
    // 3. For each follower, replicate log entries:
    //    a. Send AppendEntries starting at nextIndex[follower]
    //    b. On success, update nextIndex and matchIndex
    //    c. On failure due to log inconsistency, decrement nextIndex and retry
    // 4. Update commitIndex when entries are replicated to majority
    // 5. Apply committed entries to state machine
}

// resetElectionTimeout resets the election timer with a new random timeout.
func (node *RaftNode) resetElectionTimeout() {
    if node.electionTimeoutTimer != nil {
        node.electionTimeoutTimer.Stop()
    }
    // Add jitter to prevent simultaneous elections
    timeout := node.electionTimeout + time.Duration(rand.Intn(100))*time.Millisecond
    node.electionTimeoutTimer = time.AfterFunc(timeout, func() {
        node.mu.Lock()
        defer node.mu.Unlock()
        // TODO: Handle election timeout
    })
}
```

Language-Specific Hints for Go:

1. **Concurrency:** Use `sync.Mutex` for protecting shared state. The pattern is:

```go
node.mu.Lock()

defer node.mu.Unlock()

// Access/modify shared state
```

2. **Timeouts:** Use `time.AfterFunc` for election timeouts and `time.Ticker` for periodic heartbeats. Always stop timers when they're no longer needed to prevent goroutine leaks.
3. **Serialization:** Use struct tags with `json:"fieldName"` for JSON serialization of RPC structs. Consider using `omitempty` for optional fields.
4. **Slices and Indices:** Remember that Go slice indices start at 0, but Raft log indices typically start at 1. Use a dummy entry at index 0 to simplify index arithmetic.

Milestone Checkpoint - Section 1:

After understanding this section, you should be able to:

1. Explain the replicated state machine model in your own words.
2. Articulate why consensus is difficult from a CAP theorem perspective.
3. Contrast Raft with Paxos across at least three dimensions.
4. Set up the basic project structure with the provided HTTP transport code.
5. Understand the skeleton of the `RaftNode` struct and its main event loop.

To validate your understanding:

- Write a brief summary of the key differences between Raft and Paxos.
- Create the project directory structure and copy the HTTP transport code.
- Examine the `RaftNode` skeleton and identify where you'll implement each milestone's logic.

Debugging Tips - Foundational Issues:

Symptom	Likely Cause	How to Diagnose	Fix
"I don't understand where to start"	Overwhelmed by algorithm complexity	Break down the problem: Raft has just 3 states and 2 RPCs	Start with just the data structures, then the follower loop, then election, then replication
"My nodes can't communicate"	Transport layer misconfiguration	Check if HTTP servers are running (`netstat -tlnp`), verify addresses	Ensure each node uses a unique port and knows other nodes' addresses
"The algorithm seems too simple"	Missing edge cases	Read the Raft paper Section 5 carefully	Implement ALL rules in Figures 2 and 13, not just the happy path

2. Goals and Non-Goals

> **Milestone(s):** 1 (Leader Election), 2 (Log Replication), 3 (Safety Properties), 4 (Cluster Membership Changes)

This section defines the precise boundaries of our educational Raft implementation. In any complex software project, especially one with pedagogical objectives, explicit scoping is crucial to maintain focus, manage complexity, and ensure learners achieve core learning outcomes without being overwhelmed by production-grade concerns. We delineate what **must** be built (the goals) and what is **intentionally excluded** (the non-goals), providing clear justification for each decision.

2.1 Goals (What We Must Build)

The primary objective is to implement a **correct, understandable, and functionally complete core Raft consensus protocol** as defined in the original paper. This implementation serves as a learning vehicle for distributed systems fundamentals, with each goal directly mapping to the project's four milestones.

> **Core Learning Principle:** We prioritize **understandability over performance** and **correctness over completeness**. Every architectural choice favors clarity and educational value, even if it means implementing simpler, less optimized versions of certain mechanisms.

The following table enumerates the mandatory components and their alignment with project milestones:

Goal Component	Corresponding Milestone	Key Deliverables	Learning Objective
Leader Election	Milestone 1	• Term management and state transitions • Randomized election timeouts • `RequestVote` RPC implementation • Vote granting rules	Understand distributed failure detection, split vote resolution, and the role of terms in establishing leadership epochs
Log Replication	Milestone 2	• `AppendEntries` RPC (heartbeat and log replication) • Log consistency checks using `prevLogIndex` / `prevLogTerm` • Commit index advancement via quorum • State machine application of committed entries	Master the core replication mechanism, understand consistency guarantees, and learn how distributed logs maintain agreement
Safety Guarantees	Milestone 3	• Election safety (at most one leader per term) • Leader completeness (committed entries survive) • Log matching property • State machine safety (identical command sequences)	Internalize Raft's formal safety properties and implement the specific rules (like leader-only committing of current-term entries) that enforce them
Cluster Membership Changes	Milestone 4	• Joint consensus protocol for configuration transitions • Configuration change log entries • Safe node addition and removal procedures	Learn how to dynamically reconfigure a consensus cluster without violating safety or causing availability issues

Beyond these milestone-aligned goals, several cross-cutting architectural requirements must be satisfied:

1. **Modular Design**: The implementation must separate core Raft logic (`RaftNode`) from transport mechanisms (`HTTPServer`) and the state machine interface. This separation enables testing, reasoning about components in isolation, and swapping implementations (e.g., different network layers).
2. **Concurrency Safety**: All shared state within a `RaftNode` (particularly `currentTerm` , `votedFor` , `log` , and volatile indices) must be protected with proper synchronization (mutexes) to handle concurrent RPC requests and timer events safely.
3. **Deterministic Testing**: The system must support deterministic testing through mockable time and network interfaces, allowing reproducible validation of complex scenarios like network partitions and crash recovery.
4. **Observability**: While not production-grade, the implementation must provide sufficient logging (state transitions, RPCs received/sent) to enable debugging and understanding of the protocol's runtime behavior.

The following ADR formalizes our approach to prioritizing educational value over production features:

Decision: Educational-First Implementation Scope

- **Context**: We are building a Raft implementation primarily for learning distributed systems concepts, not for deploying in production environments. Learners need to understand the core protocol mechanics without being overwhelmed by complex infrastructure concerns.
- **Options Considered**:
  1. **Production-Ready Implementation**: Include persistent storage, optimized network layers, client libraries, and all operational features. This would provide real-world experience but would dramatically increase complexity and obscure the core protocol logic.
  2. **Pure Algorithm Simulation**: Implement only the state machine logic with no real network or storage, using simulated components for everything. This minimizes incidental complexity but provides poor intuition for real-world constraints like network latency and failure modes.
  3. **Hybrid Educational Implementation** (Chosen): Implement core protocol with real network communication and in-memory persistence, but exclude production optimizations and infrastructure. This balances understanding of real constraints with manageable complexity.
- **Decision**: Implement the hybrid educational approach with real but simplified infrastructure components.
- **Rationale**: Learners need to experience actual network communication and failure scenarios to build proper intuition, but shouldn't spend 80% of their time on infrastructure code. By providing working transport and persistence stubs, we let them focus on the consensus algorithm—the actual learning objective. The simplified components are complete enough to demonstrate real behavior but avoid deep optimization concerns.
- **Consequences**: The implementation won't be production-ready without significant additional work, but it will correctly demonstrate all Raft behaviors and safety properties. Learners can later extend it with persistent storage or better network layers as advanced exercises.

Option	Pros	Cons	Chosen?
Production-Ready Implementation	• Real-world deployable • Covers all operational concerns	• Extreme complexity obscures core algorithm • Requires extensive distributed systems experience • Long development time	❌
Pure Algorithm Simulation	• Focus purely on protocol logic • Easy to test and reason about	• Unrealistic—no real network or failures • Poor preparation for real implementation	❌
Hybrid Educational Implementation	• Real network communication demonstrates actual behavior • Manageable complexity for learners • Core algorithm remains the focus	• Not production-ready without extensions • Simplified components may misrepresent some real-world constraints	✅

2.2 Non-Goals (What We Leave Out)

To maintain focus on the core educational objectives, we explicitly exclude several production-grade features and optimizations. These are valuable topics for advanced study but would distract from understanding fundamental consensus mechanics.

Non-Goal Category	Specific Exclusions	Justification for Exclusion	Potential for Future Extension
Persistence & Crash Recovery	• Stable storage (disk) for `currentTerm`, `votedFor`, and `log` • Write-ahead logging with `fsync` operations • Snapshotting and log compaction (Raft §7)	While persistence is essential for real deployments, implementing it correctly requires deep understanding of filesystem semantics and recovery protocols. For learning, in-memory persistence is sufficient to demonstrate the algorithm's logic; crashes can be simulated via process termination.	Advanced learners can implement a `PersistentStore` interface with disk-backed storage and recovery procedures.
Production Network Layer	• Connection pooling and keep-alive • Flow control and backpressure • TLS/encryption • Sophisticated retry policies with exponential backoff	The educational focus is on RPC semantics, not network engineering. A simple HTTP/JSON transport demonstrates the message-passing paradigm without complex networking code.	The `Transport` interface allows swapping in gRPC, TCP, or other transports.
Client Interaction	• Client session management • Linearizable read optimizations (leader leases, ReadIndex) • Duplicate request detection • Client libraries in multiple languages	Client-facing concerns, while important for real systems, are separate from the core consensus algorithm. Implementing them would shift focus away from replication and agreement.	Once Raft core is complete, linearizable reads and client libraries make excellent extension projects.
Operational Features	• Dynamic reconfiguration without joint consensus (single-server changes) • Metrics collection and monitoring • Administrative APIs for cluster management • Rolling upgrade support	These features support operational maturity but don't illuminate core consensus concepts. Joint consensus, while more complex, better illustrates safety challenges during membership changes.	Metrics collection and admin APIs can be added incrementally after core functionality works.
Performance Optimizations	• Pipeline log replication (send multiple entries without waiting for responses) • Batching of small operations • Memory pooling for log entries • Zero-copy serialization	Optimizations often obscure the clear, understandable protocol flow that makes Raft pedagogically valuable. The baseline implementation should be obviously correct before optimizing.	Pipeline replication is a natural and valuable optimization to implement after understanding basic replication.
Advanced Failure Modes	• Byzantine fault tolerance (malicious nodes) • Network asymmetry (different latencies in different directions) • Clock skew beyond bounded assumptions	Raft assumes crash-stop (non-Byzantine) failures and synchronous networks with bounded delays. Handling Byzantine faults requires entirely different protocols and would derail the learning objectives.	Byzantine fault tolerance is a separate, advanced topic beyond Raft's scope.

> **Important Pedagogical Note:** These exclusions are **intentional design choices**, not omissions due to oversight. Each excluded feature represents a potential "next step" for learners who have mastered the core protocol. The implementation includes clear extension points (interfaces, pluggable components) where these features can be added later without rewriting the core algorithm.

Why In-Memory Persistence is Acceptable for Learning: While Raft requires persistent storage for safety guarantees across crashes, we can relax this requirement for educational purposes because:

1. **Simulated Crashes**: We can simulate node failures by terminating processes and restarting them with empty state, observing how the protocol recovers via term comparisons and log replication.
2. **Focus on Protocol Logic**: Disk I/O and recovery code adds significant complexity that distracts from understanding election, replication, and safety mechanisms.
3. **Verification Still Possible**: We can still verify all safety properties (no two leaders in same term, state machine safety) because these are enforced by the protocol's runtime rules, not solely by persistence.

Why Simple HTTP/JSON Transport is Sufficient: The Raft paper abstracts communication via RPCs without specifying transport details. HTTP/JSON provides:

1. **Immediate Observability**: Messages can be inspected using standard tools like `curl` or browser developer tools.
2. **Language-Neutral Interface**: JSON is widely understood and facilitates debugging.
3. **Adequate Demonstration**: It sufficiently demonstrates the asynchronous, unreliable communication model Raft assumes, including message loss (simulated by dropping requests) and reordering.

Common Pitfall: Over-Engineering Early

- ⚠️ **Pitfall: Prematurely implementing production features**
  - **Description**: Learners often want to implement snapshotting, connection pooling, or client libraries before completing the core consensus algorithm.
  - **Why It's Wrong**: This distracts from understanding the fundamental protocol and can introduce subtle bugs that obscure whether issues are in core Raft logic or ancillary features.
  - **How to Avoid**: Strictly follow the milestone progression. Complete all four core milestones with the simplified components before considering any extensions. Use the provided `Transport` and `StateMachine` interfaces as boundaries—implement the simplest working version first.

The following table summarizes the "good enough" implementation choices for our educational context:

Component	Minimal Viable Implementation	Production-Grade Alternative	Reason for Minimal Choice
Persistence	In-memory maps (lost on restart)	Disk-backed WAL with fsync	Focus on algorithm, not storage engine
Network Transport	HTTP/JSON with basic retries	gRPC with connection pooling, streaming, TLS	Debuggable, simple, demonstrates async messaging
State Machine	In-memory key-value store	Pluggable interface for any application	Concrete enough to see results of consensus
Time Management	`time.Timer` with random timeouts	Monotonic clocks with fault-tolerant timeouts	Sufficient for demonstrating timeouts and election
Concurrency Control	Single mutex protecting all `RaftNode` state	Fine-grained locking or lock-free structures	Simple correctness over performance

Implementation Guidance

Technology Recommendations Table

Component	Simple Option (Recommended)	Advanced Option (Future Extension)
Transport Layer	HTTP/JSON using Go's `net/http` package	gRPC with Protocol Buffers for efficiency
Serialization	Go's built-in `encoding/json`	Protocol Buffers or Cap'n Proto
Concurrency	`sync.Mutex` for whole `RaftNode` state	Fine-grained `sync.RWMutex` per field group
Time Management	`time.Timer` with `time.Duration`	Custom timer wheel for many nodes
Log Storage	In-memory `[]LogEntry` slice	Disk-backed append-only log with memory map
State Machine	Map-based key-value store (`map[string]string`)	Pluggable interface for custom applications

Recommended File/Module Structure:

```
raft-implementation/
├── README.md
├── go.mod
├── cmd/
│   ├── raft-node/                # Main executable for a Raft node
│   │   └── main.go
│   └── client/                   # Simple CLI client (optional)
│       └── main.go
├── internal/
│   ├── raft/                     # Core Raft algorithm implementation
│   │   ├── node.go               # RaftNode struct and main logic
│   │   ├── state.go              # NodeState, LogEntry definitions
│   │   ├── election.go           # Leader election logic (Milestone 1)
│   │   ├── replication.go        # Log replication logic (Milestone 2)
│   │   ├── safety.go             # Safety rule enforcement (Milestone 3)
│   │   ├── membership.go         # Cluster membership changes (Milestone 4)
│   │   └── raft_test.go          # Core algorithm tests
│   ├── transport/                # Network communication layer
│   │   ├── interface.go          # Transport interface definition
│   │   ├── http.go               # HTTPServer implementation
│   │   └── mock.go               # Mock transport for testing
│   ├── statemachine/             # State machine abstraction
│   │   ├── interface.go          # StateMachine interface
│   │   └── kvstore.go            # Simple key-value state machine
│   └── persistence/              # Persistence abstraction (future extension)
│       ├── interface.go
│       └── memory.go             # In-memory "persistence" for now
├── pkg/
│   └── rpc/                      # RPC message definitions
│       ├── messages.go           # RequestVote, AppendEntries structs
│       └── codec.go              # JSON serialization helpers
└── test/
    ├── scenarios/                # Integration test scenarios
    └── utilities.go              # Test utilities
```

Infrastructure Starter Code - HTTP Transport:

```go
// internal/transport/http.go

package transport

import (
    "bytes"
    "context"
    "encoding/json"
    "fmt"
    "net/http"
    "sync"
    "time"
)

// Message represents a network message between Raft nodes
type Message struct {
    Type    string          `json:"type"`    // "RequestVote", "AppendEntries", etc.
    From    int             `json:"from"`    // Sender node ID
    To      int             `json:"to"`      // Intended recipient node ID
    Payload json.RawMessage `json:"payload"` // Serialized RPC arguments
}

// Handler processes incoming messages and returns a response
type Handler func(Message) (interface{}, error)

// HTTPServer provides HTTP-based transport for Raft nodes
type HTTPServer struct {
    nodeID  int
    addr    string
    handler Handler
    server  *http.Server
    mu      sync.Mutex
    clients map[int]*http.Client // HTTP clients to other nodes
}

// NewHTTPServer creates a new HTTP transport server
func NewHTTPServer(nodeID int, addr string, handler Handler) *HTTPServer {
    return &HTTPServer{
        nodeID:  nodeID,
```

```
        addr:    addr,
        handler: handler,
        clients: make(map[int]*http.Client),
    }
}

// Start begins listening for incoming RPCs
func (s *HTTPServer) Start() error {
    mux := http.NewServeMux()
    mux.HandleFunc("/message", s.handleMessage)

    s.server = &http.Server{
        Addr:    s.addr,
        Handler: mux,
    }

    go func() {
        if err := s.server.ListenAndServe(); err != nil && err != http.ErrServerClosed {
            fmt.Printf("HTTP server error: %v\n", err)
        }
    }()

    return nil
}

// Send delivers a message to another node
func (s *HTTPServer) Send(to int, msg Message) (interface{}, error) {
    s.mu.Lock()
    client, exists := s.clients[to]
    if !exists {
        // In a real implementation, you would have a configuration mapping node IDs to addresses
        // For simplicity, we assume a naming convention like "localhost:portBase+nodeID"
        client = &http.Client{Timeout: 2 * time.Second}
        s.clients[to] = client
    }
    s.mu.Unlock()
```

```
    // Serialize message
    data, err := json.Marshal(msg)
    if err != nil {
        return nil, fmt.Errorf("failed to marshal message: %w", err)
    }

    // Determine target address (simplified - in reality, you'd have a node address map)
    targetAddr := fmt.Sprintf("http://localhost:%d/message", 8080+to)

    // Send HTTP POST
    resp, err := client.Post(targetAddr, "application/json", bytes.NewReader(data))
    if err != nil {
        return nil, fmt.Errorf("failed to send message: %w", err)
    }
    defer resp.Body.Close()

    // Decode response
    var response interface{}
    if err := json.NewDecoder(resp.Body).Decode(&response); err != nil {
        return nil, fmt.Errorf("failed to decode response: %w", err)
    }

    return response, nil
}

// Stop shuts down the HTTP server
func (s *HTTPServer) Stop() error {
    if s.server != nil {
        ctx, cancel := context.WithTimeout(context.Background(), 5*time.Second)
        defer cancel()
        return s.server.Shutdown(ctx)
    }
    return nil
}

func (s *HTTPServer) handleMessage(w http.ResponseWriter, r *http.Request) {
    if r.Method != "POST" {
```

```
        http.Error(w, "Method not allowed", http.StatusMethodNotAllowed)
        return
    }

    var msg Message
    if err := json.NewDecoder(r.Body).Decode(&msg); err != nil {
        http.Error(w, "Bad request", http.StatusBadRequest)
        return
    }

    // Process the message using the handler
    response, err := s.handler(msg)
    if err != nil {
        http.Error(w, err.Error(), http.StatusInternalServerError)
        return
    }

    // Send back the response
    w.Header().Set("Content-Type", "application/json")
    json.NewEncoder(w).Encode(response)
}
```

Core Logic Skeleton - RaftNode Structure:

GO

```go
// internal/raft/node.go

package raft

import (
    "sync"
    "time"
)

// NodeState represents the three possible states of a Raft node
type NodeState int

const (
    Follower NodeState = iota
    Candidate
    Leader
)

// LogEntry represents a single entry in the replicated log
type LogEntry struct {
    Term    int         // Term when entry was received by leader
    Index   int         // Position in the log (1-indexed)
    Command interface{} // Application command (e.g., key-value operation)
}

// RaftNode implements the core Raft consensus algorithm
type RaftNode struct {
    mu                  sync.Mutex          // Protects all shared state below

    // Persistent state on all servers (should be on stable storage)
    currentTerm         int                 // Latest term server has seen
    votedFor            int                 // CandidateId that received vote in current term (or -1)
    log                 []LogEntry          // Log entries; each entry contains command and term

    // Volatile state on all servers
    commitIndex         int                 // Index of highest log entry known to be committed
    lastApplied         int                 // Index of highest log entry applied to state machine

    // Volatile state on leaders (reinitialized after election)
```

```
    nextIndex              []int              // For each server, index of next log entry to send
    matchIndex             []int              // For each server, index of highest log entry known to be replicated

    // Node configuration
    id                     int                // Unique identifier for this node (0-indexed)
    peers                  []int              // IDs of other nodes in the cluster
    state                  NodeState          // Current state (Follower, Candidate, Leader)
    stateMachine           StateMachine       // Application state machine interface
    electionTimeout        time.Duration      // Timeout for election (randomized)
    electionTimeoutTimer   *time.Timer        // Timer for election timeout
    lastHeartbeatReceived  time.Time          // Last time a valid heartbeat was received
    transport              Transport          // Network transport interface
}

// NewRaftNode creates and initializes a new Raft node
func NewRaftNode(id int, peers []int, sm StateMachine, transport Transport) *RaftNode {
    node := &RaftNode{
        id:                 id,
        peers:              peers,
        state:              Follower,
        stateMachine:       sm,
        currentTerm:        0,
        votedFor:           -1, // -1 means no vote cast
        log:                make([]LogEntry, 0),
        commitIndex:        0,
        lastApplied:        0,
        transport:          transport,
    }

    // Initialize leader-specific arrays (will be resized when becoming leader)
    node.nextIndex = make([]int, len(peers)+1)  // +1 for simplicity in indexing
    node.matchIndex = make([]int, len(peers)+1)

    // Set initial election timeout
    node.resetElectionTimeout()
```

```go
    return node
}

// run is the main event loop for the Raft node
func (n *RaftNode) run() {
    // TODO 1: Start the main event loop
    // TODO 2: Based on current state, run the appropriate state handler:
    //   - If state == Follower: call n.runFollower()
    //   - If state == Candidate: call n.runCandidate()
    //   - If state == Leader: call n.runLeader()
    // TODO 3: Handle state transitions triggered by RPCs or timeouts
    // TODO 4: Ensure goroutine safety with proper locking
}

// resetElectionTimeout resets the election timer with a new random timeout
func (n *RaftNode) resetElectionTimeout() {
    // TODO 1: Generate a random timeout between 150-300ms (or configurable range)
    // TODO 2: Stop existing timer if it exists
    // TODO 3: Create new timer with random duration
    // TODO 4: Start timer - when it fires, transition to Candidate if still Follower
}

// runFollower handles the behavior of a node in Follower state
func (n *RaftNode) runFollower() {
    // TODO 1: Wait for either:
    //   - Election timeout (transition to Candidate)
    //   - Valid RPC from Leader or Candidate
    // TODO 2: On receiving AppendEntries from valid Leader, reset election timeout
    // TODO 3: On receiving RequestVote, process vote request
    // TODO 4: If election timeout fires, transition to Candidate
}

// runCandidate handles the behavior of a node in Candidate state
func (n *RaftNode) runCandidate() {
    // TODO 1: Increment currentTerm
    // TODO 2: Vote for self
    // TODO 3: Reset election timer
    // TODO 4: Send RequestVote RPCs to all other servers
```

```
    // TODO 5: Collect votes:
    //   - If votes from majority: become Leader
    //   - If AppendEntries from valid Leader: become Follower
    //   - If election timeout: start new election (increment term again)
    // TODO 6: Handle split vote scenario (multiple candidates)
}

// runLeader handles the behavior of a node in Leader state
func (n *RaftNode) runLeader() {
    // TODO 1: Initialize nextIndex and matchIndex for each follower
    // TODO 2: Send initial empty AppendEntries (heartbeat) to all followers
    // TODO 3: Set up periodic heartbeat timer (e.g., every 50ms)
    // TODO 4: On receiving client command: append to log, replicate to followers
    // TODO 5: Monitor responses from followers:
    //   - If successful: update matchIndex, advance commitIndex if majority replicated
    //   - If failure: decrement nextIndex and retry
    // TODO 6: Apply committed entries to state machine
    // TODO 7: If discover higher term in RPC response, transition to Follower
}
```

Language-Specific Hints for Go:

- Use `sync.Mutex` for protecting `RaftNode` state. Lock at the beginning of exported methods and release before making RPC calls to avoid deadlocks.
- Use `time.Timer` for election timeouts with `Reset()` method to reuse timers.
- When generating random timeouts, use `rand.Intn(range) + min` from the `math/rand` package, seeded with current time.
- For JSON serialization of `Command` (interface{}), you may need type assertions or custom marshaling if commands have complex structures.
- Use goroutines for handling multiple concurrent RPC requests, but ensure proper locking when accessing shared state.

Milestone Checkpoint for Section 2: After understanding the goals and non-goals, learners should:

1. **Set up the project structure** following the recommended file layout.
2. **Implement the skeleton** `RaftNode` with the basic fields and methods shown above.
3. **Test the HTTP transport** by running two nodes that can send messages to each other:

```
# In one terminal
go run cmd/raft-node/main.go --id=1 --peers=2,3 --port=8081

# In another terminal
go run cmd/raft-node/main.go --id=2 --peers=1,3 --port=8082
```

Verify that nodes can communicate by checking log output for successful HTTP requests. 4. **Expected sign of progress**: No consensus behavior yet, but nodes should start up without errors and be able to exchange basic messages via the transport layer.

Debugging Tips for Early Setup:

Symptom	Likely Cause	How to Diagnose	Fix
"Connection refused" errors	Wrong port numbers or nodes not started	Check that each node is listening on the expected port with `netstat -tulpn`	Update port configuration or startup order
JSON unmarshaling errors	Incompatible message structures between nodes	Log the raw JSON being sent/received	Ensure both nodes use identical `Message` struct definitions
Deadlock on startup	Mutex not released before network call	Use a debugger or add logging before/after locks	Release mutex before making RPC calls via `transport.Send()`

3. High-Level Architecture

> **Milestone(s):** 1 (Leader Election), 2 (Log Replication), 3 (Safety Properties), 4 (Cluster Membership Changes)

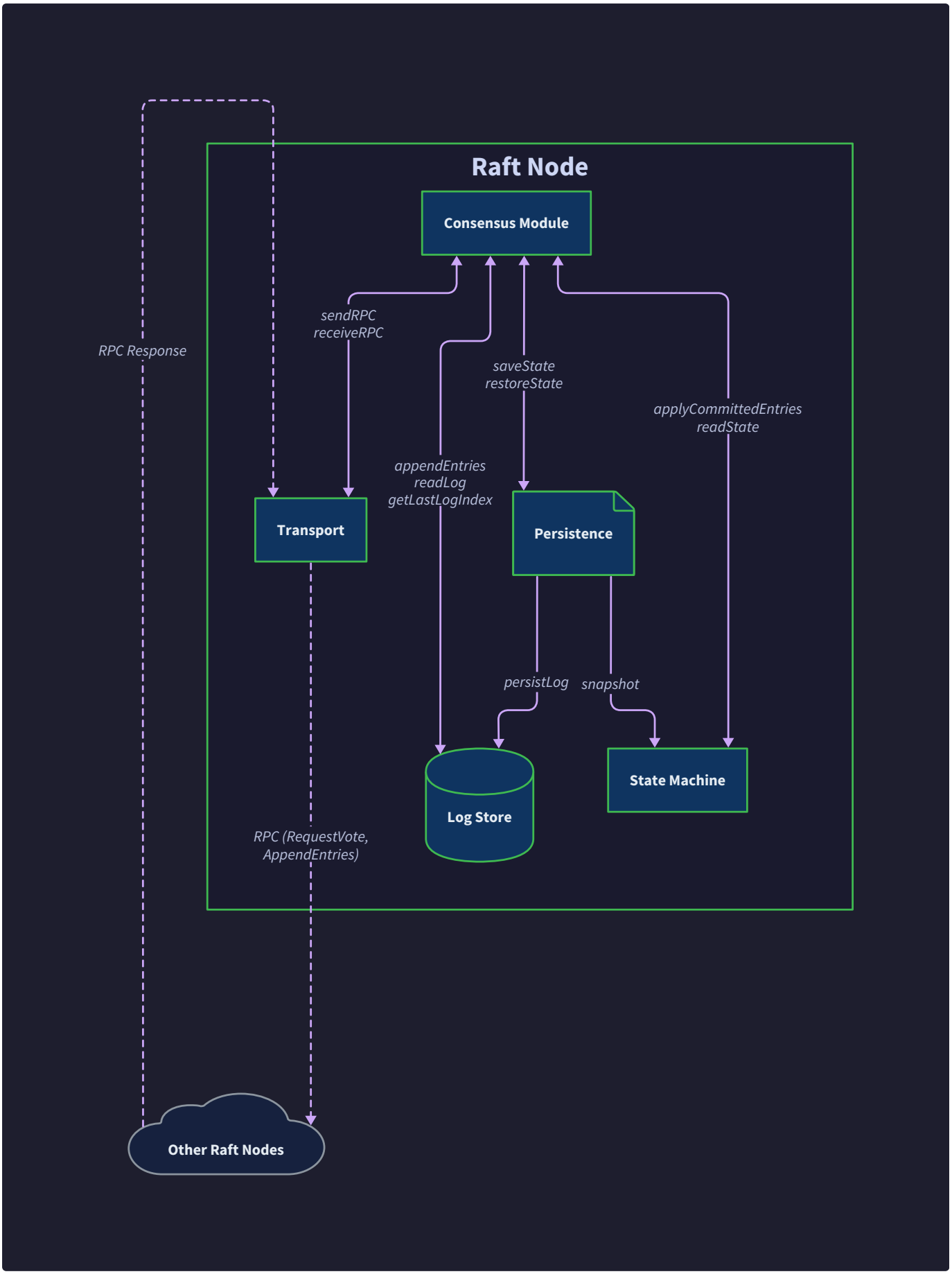
This section provides the blueprint for constructing a Raft node—the fundamental building block of our distributed consensus system. Before diving into protocol details, we must establish a clear mental picture of how the components fit together and how the codebase should be organized. A well-structured architecture makes the complex state machine manageable and the implementation journey more navigable.

3.1 Component Overview

Think of a Raft node as a **three-person team running a highly secure document processing office**:

1. **The Manager (Consensus Module)** makes decisions about who's in charge and what work gets approved
2. **The Archivist (Log Store)** maintains the master ledger of all work orders in sequence
3. **The Worker (State Machine)** actually performs the work described in approved orders
4. **The Messenger (Transport)** delivers memos between different offices (nodes)
5. **The Safe (Persistence)** stores critical records that must survive power outages

This division of responsibilities ensures each component has a single, well-defined purpose, making the system easier to reason about and test.


Raft Node
Consensus Module
sendRPC
receiveRPC
RPC Response
saveState
restoreState
applyCommittedEntries
readState
appendEntries
readLog
getLastLogIndex
Transport
Persistence
persistLog
snapshot
Log Store
State Machine
RPC (RequestVote,
AppendEntries)
Other Raft Nodes

3.1.1 Consensus Module: The Decision Maker

The **Consensus Module** is the brain of the Raft node, implementing the core state machine logic described in the Raft paper. It orchestrates all other components and manages the node's role transitions between `Follower`, `Candidate`, and `Leader` states. This module directly corresponds to the `RaftNode` struct in our implementation.

Responsibility	Description	Key Data Managed
State Management	Maintains current role (`NodeState`), term number, and voted-for candidate	`currentTerm`, `votedFor`, `state`
Leader Election	Initiates elections, requests votes, and handles voting decisions	`electionTimeoutTimer`, `lastHeartbeatReceived`
Log Replication	Sends `AppendEntries` RPCs as leader, processes them as follower	`nextIndex`, `matchIndex`, `commitIndex`
Safety Enforcement	Ensures election safety and log matching properties through protocol rules	Term comparison logic, log completeness checks
Timing Coordination	Manages election timeouts and heartbeat intervals to detect failures	`electionTimeout`, timer reset logic

The Consensus Module follows a reactive design pattern: it waits for events (timer expirations, incoming RPCs, client requests) and responds by updating internal state and potentially sending messages to other nodes.

> **Architecture Insight:** The Consensus Module is intentionally kept free of networking and persistence details. This separation allows us to test the core consensus logic in isolation using mock transports and in-memory storage, which is crucial for building confidence in the implementation's correctness.

3.1.2 Log Store: The Immutable Ledger

The **Log Store** maintains the sequence of commands that have been proposed to the cluster. It's more than just storage—it's a critical component for ensuring the **Log Matching Property** (if two logs contain an entry with the same index and term, they are identical up to that index).

Responsibility	Description	Implementation Considerations
Entry Storage	Stores log entries with their term, index, and command payload	Array-like structure with O(1) append and index access
Consistency Checks	Verifies log prefixes match during replication (`prevLogIndex`, `prevLogTerm`)	Fast term lookup by index, boundary checking
Truncation Support	Removes conflicting entries when leader provides different entries at same index	Safe slice manipulation, memory management
Persistence Hook	Provides durability interface for critical log entries	Sync-to-disk coordination, checkpointing

The log is conceptually an array that starts at index 1 (not 0). Each entry contains three fields:

- `Term`: The leader's term when the entry was created
- `Index`: The position in the log sequence
- `Command`: The application-specific operation to execute

> **Critical Design Principle:** The log is **append-only** except for truncation during conflict resolution. Once an entry is committed (replicated to a majority), it can never be removed or modified—this immutability is fundamental to Raft's safety guarantees.

3.1.3 State Machine: The Application Logic

The **State Machine** represents the application built on top of Raft—for example, a key-value store, configuration manager, or database. It's a separate component that the Consensus Module notifies when log entries become committed and ready for execution.

Responsibility	Description	Interface Pattern
Command Application	Executes committed commands in log order	`Apply(Command) (Result, error)`
State Isolation	Maintains application state separate from consensus state	Clear separation of concerns
Determinism Guarantee	Produces identical results given the same command sequence on all nodes	Pure functions, no external dependencies
Result Reporting	Returns results to clients via leader (optional)	Async callback or channel notification

The State Machine must be **deterministic**: given the same sequence of commands, every replica must arrive at the same final state. Non-deterministic operations (like random number generation or reading the system clock) must be carefully controlled or avoided.

3.1.4 Transport: The Communication Layer

The **Transport** component abstracts away the networking details, allowing the Consensus Module to send and receive RPCs without worrying about serialization, connection management, or retry logic.

Responsibility	Description	Implementation Options
Message Delivery	Sends RPCs to other nodes and receives incoming requests	HTTP/gRPC/TCP with JSON/Protobuf
Serialization	Converts Go structs to wire format and back	JSON, Protocol Buffers, custom binary
Connection Management	Maintains persistent connections or creates on-demand	Connection pooling, keep-alives
Failure Masking	Handles network timeouts, retries, and transient failures	Exponential backoff, circuit breakers

The Transport interface defines two critical operations:

1. `Send(to int, msg Message) (interface{}, error)`: Deliver a message to a specific peer
2. Registration for incoming message handlers (callback pattern)

> **Decision: Abstract Transport Interface**
>
> - **Context**: The core Raft algorithm should be testable without real network communication, and different deployments may prefer different networking stacks (HTTP for simplicity, gRPC for performance).
> - **Options Considered**:
>   1. **Direct HTTP client in Consensus Module**: Hardcode HTTP calls throughout the Raft logic
>   2. **Abstract interface with default HTTP implementation**: Define `Transport` interface, provide working HTTP implementation
>   3. **Event-based messaging bus**: Use channels and async events for all communication
> - **Decision**: Abstract interface with default HTTP implementation (Option 2)
> - **Rationale**: This keeps the consensus logic cleanly separated from networking concerns, enables easy mocking for tests, and allows swapping implementations without changing the core algorithm. The HTTP default provides a simple, standards-based option that works out of the box.
> - **Consequences**: Adds a small layer of indirection but pays off in testability and flexibility. Learners can focus on Raft first, then optimize networking later.

Option	Pros	Cons	Chosen?
Direct HTTP in Consensus	Simple to implement initially	Hard to test, tightly couples networking to algorithm, difficult to change later	❌
Abstract Transport Interface	Testable, swappable implementations, clean separation	Slight complexity increase, interface design required	✅
Event-based Messaging Bus	Highly decoupled, good for simulation	More complex, further from Raft paper's RPC model	❌

3.1.5 Persistence: The Crash Survival Mechanism

Persistence ensures that certain critical state survives node crashes and restarts. Without persistence, a restarted node could violate safety guarantees by forgetting previous decisions.

Data to Persist	Why It Must Survive Crashes	When to Write
`currentTerm`	Prevents term regression which could lead to multiple leaders in same term	After any term change
`votedFor`	Prevents double-voting in same term (violates election safety)	After voting decision
`log[]`	Preserves committed entries; losing them breaks state machine safety	After log append (can batch)

The persistence component typically writes to disk, but for the learning implementation, we can start with an in-memory store that simulates persistence, then add actual file I/O as an extension.

> **Important Safety Note**: The Raft paper requires that updates to these persistent state variables be written to stable storage **before** responding to RPCs. In practice, this means calling `fsync()` or its equivalent. For learning purposes, we'll model this requirement but may defer actual disk persistence initially.

3.2 Recommended File/Module Structure

A clean, logical code structure is essential for managing the complexity of a Raft implementation. The following layout separates concerns, makes testing easier, and follows Go conventions.

3.2.1 Project Layout Philosophy

Our structure follows these principles:

1. **Separation by responsibility**: Each major component lives in its own package
2. **Testability**: Interfaces allow mocking of dependencies
3. **Progressive disclosure**: Simple implementations first, optimizations later
4. **Clear entry points**: Easy to find where execution begins

Here's the recommended directory structure:

```
build-your-own-raft/
├── cmd/
│   ├── raft-node/                  # Main executable for a Raft node
│   │   └── main.go                 # CLI parsing, node initialization
│   └── client/                     # Optional: test client
│       └── main.go                 # Sends commands to cluster
├── internal/                       # Private application code
│   ├── consensus/                  # Core Raft algorithm (Milestones 1-4)
│   │   ├── raft.go                 # RaftNode struct and main logic
│   │   ├── state.go                # NodeState, LogEntry definitions
│   │   ├── election.go             # Leader election logic (Milestone 1)
│   │   ├── replication.go          # Log replication logic (Milestone 2)
│   │   ├── safety.go               # Safety enforcement (Milestone 3)
│   │   ├── membership.go           # Cluster changes (Milestone 4)
│   │   └── consensus_test.go       # Unit tests for consensus logic
│   ├── storage/                    # Log and persistence layer
│   │   ├── logstore.go             # Log storage interface and implementation
│   │   ├── memory_store.go         # In-memory implementation (starter)
│   │   ├── persistent_store.go     # Disk-based implementation (extension)
│   │   └── storage_test.go
│   ├── transport/                  # Networking abstraction
│   │   ├── transport.go            # Transport interface
│   │   ├── http_transport.go       # HTTP implementation (starter)
│   │   ├── mock_transport.go       # Mock for testing
│   │   └── transport_test.go
│   └── statemachine/               # Application state machine
│       ├── interface.go            # StateMachine interface
│       ├── kv_store.go             # Example: key-value store
│       └── mock_sm.go              # Mock for testing
├── pkg/                            # Public libraries (if any)
│   └── protocol/                   # Shared message formats
│       ├── messages.go             # RPC structs (RequestVote, AppendEntries)
│       └── types.go                # Common types
├── scripts/                        # Helper scripts
│   ├── start-cluster.sh            # Start a 3-node cluster
│   └── test-partition.sh           # Simulate network partition
├── go.mod                         # Go module definition
└── README.md                      # Project documentation
```

3.2.2 Package Responsibilities and Dependencies

The dependency flow should be unidirectional to avoid circular imports:

```
raft-node (cmd) → consensus → storage → (no dependency on consensus)
                    ↓
                transport (interface)
                    ↓
              http_transport (implementation)
```

Consensus Package (`internal/consensus/`)

- **Purpose**: Implements the core Raft state machine
- **Depends on**: `storage` (interface), `transport` (interface), `statemachine` (interface)
- **Key files**:
  - `raft.go` : Contains the `RaftNode` struct and main event loop (`run()` method)
  - `state.go` : `NodeState` enum, `LogEntry` struct, and other type definitions
  - `election.go` : `runCandidate()` , `requestVote()` , `handleRequestVoteRPC()`
  - `replication.go` : `runLeader()` , `appendEntries()` , `handleAppendEntriesRPC()`
  - `safety.go` : `advanceCommitIndex()` , safety rule checks
  - `membership.go` : Joint consensus logic for configuration changes

Storage Package (`internal/storage/`)

- **Purpose**: Manages log storage and persistence
- **Depends on**: Nothing (low-level package)

- **Key files**:
  - `logstore.go`: `LogStore` interface with methods like `Append()`, `Get()`, `Truncate()`
  - `memory_store.go`: Simple in-memory implementation for initial development
  - `persistent_store.go`: Disk-backed implementation with crash safety (advanced)

Transport Package (`internal/transport/`)

- **Purpose**: Abstracts network communication
- **Depends on**: `protocol` (for message types)
- **Key files**:
  - `transport.go`: `Transport` interface with `Send()` and `Start()` methods
  - `http_transport.go`: `HTTPServer` struct implementing the interface using HTTP/JSON
  - `mock_transport.go`: In-memory transport for unit tests

StateMachine Package (`internal/statemachine/`)

- **Purpose**: Application logic that executes committed commands
- **Depends on**: Nothing (application-specific)
- **Key files**:
  - `interface.go`: `StateMachine` interface with `Apply(command interface{})` method
  - `kv_store.go`: Example implementation: in-memory key-value store
  - `mock_sm.go`: No-op implementation for testing consensus without application logic

Protocol Package (`pkg/protocol/`)

- **Purpose**: Shared message formats between nodes
- **Depends on**: Nothing (just data structures)
- **Key files**:
  - `messages.go`: `RequestVoteArgs`, `AppendEntriesArgs`, and their response structs
  - `types.go`: Common types used across packages

> **Implementation Strategy:** Start with the simplest possible implementation in each package (in-memory storage, mock transport) to get the consensus logic working, then replace with more robust implementations. This "walk before you run" approach prevents getting bogged down in peripheral complexities before the core algorithm is correct.

3.2.3 Interface Definitions for Loose Coupling

To enable testing and flexible implementation swapping, we define these key interfaces:

Interface	Location	Purpose	Key Methods
`LogStore`	`internal/storage/logstore.go`	Abstract log storage	`Append(entries []LogEntry)`, `Get(index int)`, `LastIndex()`, `Truncate(index int)`
`Transport`	`internal/transport/transport.go`	Abstract network communication	`Send(to int, msg Message)`, `Start()`, `Stop()`, `RegisterHandler(handler Handler)`
`StateMachine`	`internal/statemachine/interface.go`	Abstract application	`Apply(command interface{})`, `GetState() interface{}`
`PersistentStore`	`internal/storage/logstore.go`	Abstract persistence	`SaveTerm(term int)`, `SaveVotedFor(candidateId int)`, `SaveLog(entries []LogEntry)`

These interfaces allow us to write the consensus logic against abstractions, then provide multiple implementations (in-memory for testing, disk-based for production, HTTP for communication, in-process channels for simulation).

Implementation Guidance

This section bridges the architectural design to concrete code, providing starter implementations for infrastructure components and skeleton code for the core Raft logic.

A. Technology Recommendations

Component	Simple Option (Recommended for Learning)	Advanced Option (For Production)
Transport	HTTP/1.1 with JSON (Go's `net/http`)	gRPC with Protocol Buffers
Serialization	JSON (`encoding/json`)	Protocol Buffers (`google.golang.org/protobuf`)
Persistence	In-memory first, then append-only log file	WAL with batch fsync, snapshotting
Concurrency	Goroutines + mutexes (`sync.Mutex`)	More fine-grained locking or actor model
Testing	Go's testing package + mocks	Property-based tests (`gopter`), fault injection

B. Starter Infrastructure Code

File: `internal/transport/http_transport.go`

```go
package transport

import (
    "encoding/json"
    "fmt"
    "net/http"
    "sync"
    "time"
)

// Message represents a network message between Raft nodes
type Message struct {
    Type    string          `json:"type"`    // "RequestVote", "AppendEntries", etc.
    From    int             `json:"from"`    // Sender node ID
    To      int             `json:"to"`      // Intended recipient (for routing)
    Payload json.RawMessage `json:"payload"` // Serialized RPC args
}

// Handler processes incoming messages and returns a response
type Handler func(Message) (interface{}, error)

// HTTPServer implements Transport using HTTP/JSON
type HTTPServer struct {
    nodeID   int
    addr     string
    handler  Handler
    server   *http.Server
    mu       sync.Mutex
    clients  map[int]*http.Client // HTTP clients to other nodes
}

// NewHTTPServer creates a new HTTP transport server
func NewHTTPServer(nodeID int, addr string, handler Handler) *HTTPServer {
    return &HTTPServer{
        nodeID:  nodeID,
        addr:    addr,
        handler: handler,
        clients: make(map[int]*http.Client),
```

```
    }
}

// Start begins listening for incoming RPCs
func (h *HTTPServer) Start() error {
    mux := http.NewServeMux()
    mux.HandleFunc("/message", h.handleMessage)

    h.server = &http.Server{
        Addr:    h.addr,
        Handler: mux,
    }

    go func() {
        if err := h.server.ListenAndServe(); err != nil && err != http.ErrServerClosed {
            fmt.Printf("HTTP server error: %v\n", err)
        }
    }()

    return nil
}

// Send delivers a message to another node
func (h *HTTPServer) Send(to int, msg Message) (interface{}, error) {
    h.mu.Lock()
    client, exists := h.clients[to]
    if !exists {
        // In reality, you'd have a map of nodeID->address
        // For simplicity, we assume localhost with different ports
        client = &http.Client{Timeout: 5 * time.Second}
        h.clients[to] = client
    }
    h.mu.Unlock()

    // Serialize message
    data, err := json.Marshal(msg)
    if err != nil {
```

```
        return nil, err
    }

    // Send HTTP POST (in reality, you'd know the target address)
    // This is simplified - you'd need to map nodeID to actual address
    resp, err := client.Post(fmt.Sprintf("http://localhost:%d/message", 8000+to),
                             "application/json",
                             bytes.NewReader(data))
    if err != nil {
        return nil, err
    }
    defer resp.Body.Close()

    // Parse response
    var response interface{}
    if err := json.NewDecoder(resp.Body).Decode(&response); err != nil {
        return nil, err
    }

    return response, nil
}

// Stop shuts down the HTTP server
func (h *HTTPServer) Stop() error {
    if h.server != nil {
        return h.server.Shutdown(context.Background())
    }
    return nil
}

func (h *HTTPServer) handleMessage(w http.ResponseWriter, r *http.Request) {
    var msg Message
    if err := json.NewDecoder(r.Body).Decode(&msg); err != nil {
        http.Error(w, err.Error(), http.StatusBadRequest)
        return
    }
```

```
    // Process the message through the handler
    response, err := h.handler(msg)
    if err != nil {
        http.Error(w, err.Error(), http.StatusInternalServerError)
        return
    }

    // Send back the response
    w.Header().Set("Content-Type", "application/json")
    json.NewEncoder(w).Encode(response)
}
```

File: `internal/storage/memory_store.go`

```go
package storage

import (
    "sync"
)

// LogEntry represents a single entry in the Raft log
type LogEntry struct {
    Term    int         `json:"term"`
    Index   int         `json:"index"`
    Command interface{} `json:"command"`
}

// MemoryStore is an in-memory implementation of LogStore
type MemoryStore struct {
    mu        sync.RWMutex
    entries   []LogEntry
    // In a real implementation, you'd also persist currentTerm and votedFor
    currentTerm int
    votedFor    int
}

// NewMemoryStore creates a new in-memory log store
func NewMemoryStore() *MemoryStore {
    // Start with a dummy entry at index 0 for easier indexing
    return &MemoryStore{
        entries: []LogEntry{{Term: 0, Index: 0, Command: nil}},
    }
}

// Append adds entries to the log
func (m *MemoryStore) Append(entries []LogEntry) error {
    m.mu.Lock()
    defer m.mu.Unlock()

    for _, entry := range entries {
        // Basic validation
        if entry.Index != len(m.entries) {
```

```
            return fmt.Errorf("log index mismatch: expected %d, got %d",
                              len(m.entries), entry.Index)
        }
        m.entries = append(m.entries, entry)
    }

    return nil
}

// Get returns the entry at the specified index
func (m *MemoryStore) Get(index int) (LogEntry, error) {
    m.mu.RLock()
    defer m.mu.RUnlock()

    if index < 0 || index >= len(m.entries) {
        return LogEntry{}, fmt.Errorf("index %d out of bounds", index)
    }

    return m.entries[index], nil
}

// LastIndex returns the index of the last entry in the log
func (m *MemoryStore) LastIndex() int {
    m.mu.RLock()
    defer m.mu.RUnlock()
    return len(m.entries) - 1
}

// GetTerm returns the term of the entry at the given index
func (m *MemoryStore) GetTerm(index int) (int, error) {
    entry, err := m.Get(index)
    if err != nil {
        return 0, err
    }
    return entry.Term, nil
}

// Truncate removes all entries from the given index onward
```

```go
func (m *MemoryStore) Truncate(fromIndex int) error {
    m.mu.Lock()
    defer m.mu.Unlock()

    if fromIndex < 1 || fromIndex > len(m.entries) {
        return fmt.Errorf("truncate index %d out of bounds", fromIndex)
    }

    m.entries = m.entries[:fromIndex]
    return nil
}

// Slice returns a slice of entries between start and end indices
func (m *MemoryStore) Slice(start, end int) ([]LogEntry, error) {
    m.mu.RLock()
    defer m.mu.RUnlock()

    if start < 0 || end > len(m.entries) || start > end {
        return nil, fmt.Errorf("invalid slice range [%d, %d)", start, end)
    }

    result := make([]LogEntry, end-start)
    copy(result, m.entries[start:end])
    return result, nil
}
```

C. Core Raft Node Skeleton

File: `internal/consensus/raft.go`

```go
package consensus

import (
    "fmt"
    "sync"
    "time"

    "github.com/your-username/raft/internal/storage"
    "github.com/your-username/raft/internal/transport"
    "github.com/your-username/raft/internal/statemachine"
)

// NodeState represents the possible states of a Raft node
type NodeState int

const (
    Follower NodeState = iota
    Candidate
    Leader
)

// RaftNode implements the core Raft consensus algorithm
type RaftNode struct {
    // Persistent state (must survive crashes)
    mu          sync.Mutex
    currentTerm int
    votedFor    int            // candidateId that received vote in current term
    log         []LogEntry     // In practice, use storage.LogStore interface

    // Volatile state on all servers
    commitIndex int
    lastApplied int

    // Volatile state on leaders (reinitialized after election)
    nextIndex   []int
    matchIndex  []int

    // Node identity and cluster information
```

```
    id          int
    peers       []int          // IDs of other nodes in cluster

    // Current state
    state       NodeState

    // Component dependencies
    stateMachine statemachine.StateMachine
    logStore     storage.LogStore
    transport    transport.Transport

    // Timing
    electionTimeout      time.Duration
    electionTimeoutTimer *time.Timer
    lastHeartbeatReceived time.Time

    // Channels for event-driven design (alternative approach)
    // requestVoteCh   chan RequestVoteArgs
    // appendEntriesCh chan AppendEntriesArgs
    // applyCh         chan ApplyMsg
}

// NewRaftNode creates and initializes a new Raft node
func NewRaftNode(id int, peers []int, sm statemachine.StateMachine,
                 transport transport.Transport) *RaftNode {
    node := &RaftNode{
        id:           id,
        peers:        peers,
        stateMachine: sm,
        transport:    transport,
        state:        Follower,
        currentTerm:  0,
        votedFor:     -1, // -1 means no vote cast
        commitIndex:  0,
        lastApplied:  0,
        // Initialize with empty log (index 0 is dummy entry)
```

```
        log:          []LogEntry{{Term: 0, Index: 0, Command: nil}},
    }

    // Set randomized election timeout (150-300ms typical for learning)
    node.electionTimeout = time.Duration(150 + rand.Intn(150)) * time.Millisecond
    node.electionTimeoutTimer = time.NewTimer(node.electionTimeout)

    // Initialize leader state arrays (will be resized when becoming leader)
    node.nextIndex = make([]int, len(peers)+1)  // +1 for 1-indexed simplicity
    node.matchIndex = make([]int, len(peers)+1)

    return node
}

// run is the main event loop for the Raft node
func (rn *RaftNode) run() {
    // TODO 1: Start by running as a follower
    // TODO 2: Enter main loop that checks for state transitions
    // TODO 3: In each iteration:
    //   - Check election timeout (if follower/candidate)
    //   - Send heartbeats (if leader)
    //   - Process incoming RPCs via transport callbacks
    //   - Apply committed entries to state machine
    //   - Handle state transitions (follower -> candidate -> leader -> follower)
    // TODO 4: Ensure all state transitions are protected by mutex
    // TODO 5: Implement graceful shutdown signal handling

    rn.runFollower() // Start in follower state
}

// runFollower handles the behavior of a node in Follower state
func (rn *RaftNode) runFollower() {
    // TODO 1: Reset election timer with random timeout
    // TODO 2: Loop while state == Follower
    // TODO 3: In loop:
    //   - Wait for either election timeout or incoming RPC
    //   - If election timeout: transition to Candidate state
```

```
    //   - If AppendEntries RPC with term >= currentTerm: reset timer
    //   - If RequestVote RPC: process voting request
    //   - If term in any RPC > currentTerm: update term and revert to follower
    // TODO 4: Implement proper channel-based waiting (select statement)
    // TODO 5: Ensure timer is properly stopped/reset on state exit
}

// runCandidate handles the behavior of a node in Candidate state
func (rn *RaftNode) runCandidate() {
    // TODO 1: Increment currentTerm
    // TODO 2: Vote for self
    // TODO 3: Reset election timer
    // TODO 4: Send RequestVote RPCs to all peers
    // TODO 5: Collect votes in goroutine-safe manner
    // TODO 6: If votes received from majority: become Leader
    // TODO 7: If AppendEntries RPC received from valid leader: become Follower
    // TODO 8: If election timeout: start new election (increment term again)
    // TODO 9: If discover higher term in any RPC: revert to Follower
}

// runLeader handles the behavior of a node in Leader state
func (rn *RaftNode) runLeader() {
    // TODO 1: Initialize nextIndex and matchIndex for each follower
    // TODO 2: Start sending periodic heartbeats (AppendEntries with no entries)
    // TODO 3: For each follower, start a replication goroutine
    // TODO 4: Handle client requests: append to log, replicate to followers
    // TODO 5: Track replication progress and advance commitIndex
    // TODO 6: If discover higher term: revert to Follower
    // TODO 7: Implement flow control to avoid overwhelming followers
}

// resetElectionTimeout resets the election timer with a new random timeout
func (rn *RaftNode) resetElectionTimeout() {
    // TODO 1: Stop existing timer if running
    // TODO 2: Generate new random timeout in configured range
    // TODO 3: Start new timer with that duration
    // TODO 4: Handle timer channel in the appropriate state loop
```

```
}
```

D. Language-Specific Hints for Go

1. **Concurrency Model**: Use `sync.Mutex` for protecting shared state in `RaftNode`. Be careful to avoid holding locks while making RPC calls (network I/O) to prevent deadlocks.
2. **Timer Management**: Go's `time.Timer` can be tricky—always check the return value of `Stop()` and drain the channel if needed. Consider using `time.NewTimer` and resetting it rather than creating new timers.
3. **Interface Design**: Define small, focused interfaces. For example:

```go
type LogStore interface {
    Append(entries []LogEntry) error
    Get(index int) (LogEntry, error)
    LastIndex() int
    GetTerm(index int) (int, error)
}
```

4. **Error Handling**: Use Go's multiple return values for errors. For RPC responses, include both a success bool and an error for network failures.
5. **Testing**: Use interface mocks extensively. The `transport.Transport` and `storage.LogStore` interfaces enable testing the consensus logic without real networking or disk I/O.
6. **JSON Serialization**: When using `encoding/json` for RPCs, be aware that interface{} types need special handling. Consider using concrete types or custom marshalers.
7. **Goroutine Lifecycle**: Keep track of spawned goroutines (for replication to each follower) and ensure they're properly cleaned up on state transitions.
8. **Channel Patterns**: While not required, channels can simplify the event-driven design. Consider having separate channels for different event types (timer expirations, RPC requests, client commands).

---

4. Data Model

> **Milestone(s):** 1 (Leader Election), 2 (Log Replication), 3 (Safety Properties), 4 (Cluster Membership Changes)

The data model forms the core vocabulary of the Raft protocol. Like a legal contract that defines exact terms and conditions, every field in these structures has precise semantics that collectively ensure the system's correctness. A single misinterpretation—such as treating a `term` as a simple sequence number rather than a leadership epoch—can break the entire consensus algorithm. This section defines the foundational data structures that will appear in every component of your implementation, establishing a shared language for the subsequent design discussions.

4.1 Persistent State

> **Mental Model: The Captain's Log on a Ship** Imagine a ship's captain maintains a formal logbook that records two critical pieces of information: (1) the current voyage number (which increases with each new captain) and (2) who they voted for as captain during the last crew election. This logbook is stored in a waterproof safe that survives storms. Even if the ship is wrecked and rebuilt, the new crew can open the safe, read the last recorded voyage number and vote, and ensure they don't accidentally revert to an earlier, less informed state. This is exactly what persistent state provides in Raft—it survives server crashes and restarts, preventing "amnesia" that could cause safety violations.

Persistent state must be written to stable storage (like disk) before a server responds to certain RPCs and must survive crashes. The Raft paper identifies three pieces that must be persisted:

1. `currentTerm`: The latest term this server has seen (initialized to 0 on first boot, increases monotonically)

2. **`votedFor`** : The candidate ID this server voted for in the current term (or `nil` if none)
3. **`log[]`** : The sequence of log entries, each containing a command for the state machine, the term when the entry was created, and its index in the log

These three elements form what the Raft paper calls the "critical triad" for election safety. Without persisting them, a crashed server could reboot and:

- Forget it already voted in a term, voting twice and creating multiple leaders
- Revert to an older term, violating the monotonicity guarantee
- Lose committed log entries, breaking the state machine safety property

> **Design Insight:** The persistence requirement follows a simple rule: *any state that influences future voting or log acceptance decisions must survive crashes*. If you're uncertain whether a field needs persistence, ask: "If this value were lost after a crash, could the server make a different decision than it would have made before the crash?" If yes, it must be persistent.

The following table defines the exact persistent state fields as they appear in our `RaftNode` and `MemoryStore` structures:

Field Name	Type	Description	Persistence Rationale
`currentTerm`	`int`	Latest term server has seen (monotonically increasing).	Prevents server from reverting to older term after crash, which could cause it to accept stale leader RPCs or vote for candidates with outdated logs.
`votedFor`	`int`	Candidate ID that received vote in current term (or -1/null if none).	Prevents double-voting within same term after crash, which could lead to split votes and multiple leaders.
`log[]`	`[]LogEntry`	Log entries; each entry contains command for state machine, and term when entry was received by leader (first index is 1).	Ensures committed entries survive crashes. Without this, a leader could forget about committed entries and allow them to be overwritten, violating State Machine Safety.

Each `LogEntry` in the log has its own internal structure:

Field Name	Type	Description	Purpose in Protocol
`Term`	`int`	Term when entry was created by leader.	Used for log consistency checks: followers compare terms to detect conflicting entries and ensure Leader Completeness property.
`Index`	`int`	Position in log (1-indexed).	Provides absolute ordering; ensures all servers apply commands in same sequence.
`Command`	`interface{}`	Command to apply to state machine (opaque to Raft).	The actual payload that the replicated state machine will execute once the entry is committed.

ADR: Storing Log Index vs. Computing It Implicitly

Context: We need to associate each log entry with its position in the log sequence. The index is crucial for the AppendEntries consistency check (`prevLogIndex`) and for tracking replication progress.

Options Considered:

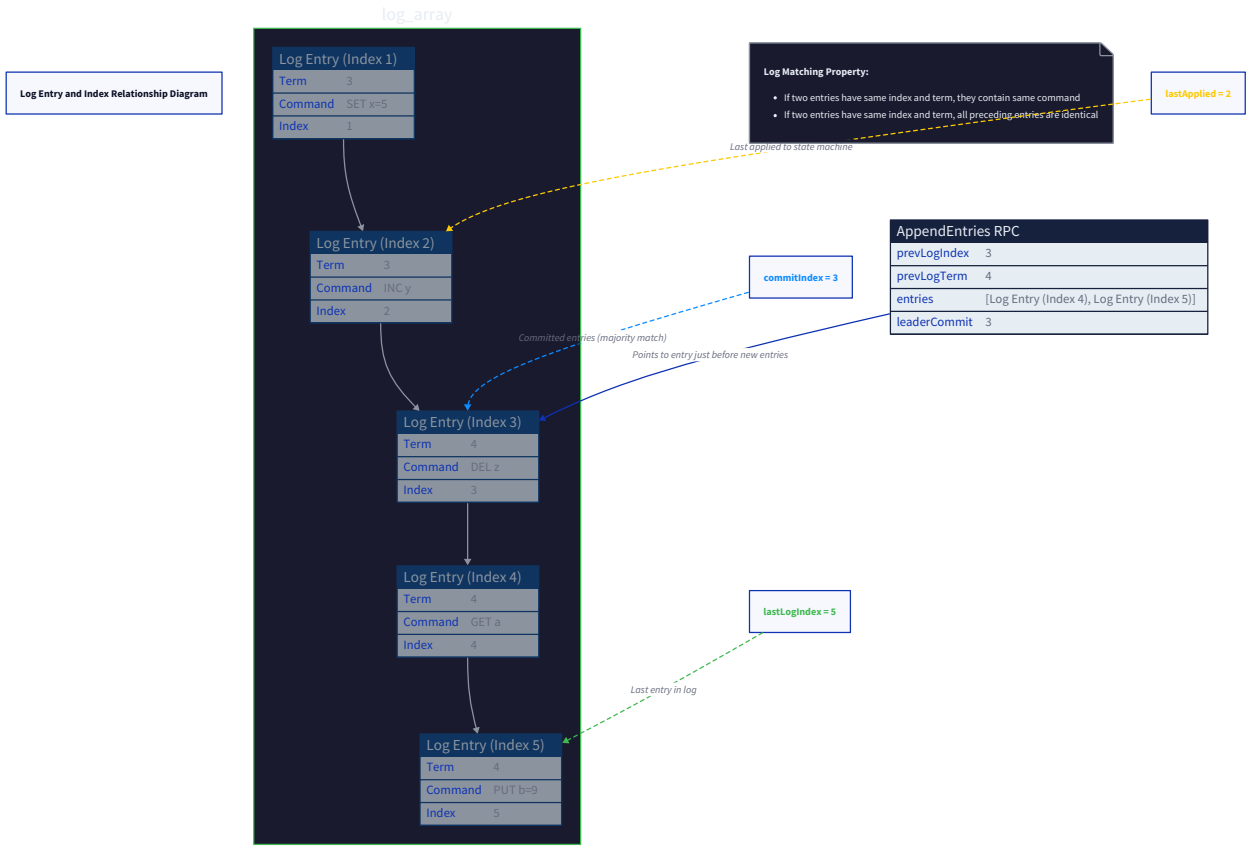
1. **Store index explicitly in each entry** (as shown above)
2. **Compute index implicitly from array position** (entry at `log[i]` has index `i+1`)

Option	Pros	Cons
Explicit index field	- Self-contained entries - Easier debugging (entry shows its own index) - Simplifies serialization/deserialization	- Storage overhead (extra integer per entry) - Potential for inconsistency if index doesn't match position
Implicit from position	- No storage overhead - Impossible to have index/position mismatch	- Requires careful handling during log truncation/compaction - Debugging requires knowing array position - Serialization must preserve order

Decision: Use implicit indexing (array position) for the actual implementation, but include Index field in the `LogEntry` type for clarity in documentation and debugging output.

Rationale: The Raft paper describes the log as an array indexed from 1, where the entry's position implicitly defines its index. Storing redundant information invites bugs where the two values could diverge. However, for debugging and clarity in our educational implementation, we'll keep the field but ensure it's always consistent with the entry's position.

Consequences: We must be meticulous when manipulating the log array (slicing, appending) to maintain the invariant: `log[i].Index == i+1` . The `LastIndex()` helper method will return `len(log)` rather than examining the last entry's Index field.



4.2 Volatile State

> **Mental Model: The Construction Foreman's Clipboard** Imagine a construction foreman (leader) with a clipboard tracking: (1) which work items are finalized (`commitIndex`), (2) which have been handed to workers (`lastApplied`), (3) where each worker should start next task (`nextIndex`), and (4) how far each worker has confirmed (`matchIndex`). This clipboard is useful for daily operations but isn't archived—if the foreman loses it, they can reconstruct it by asking workers about their progress. Similarly, Raft's volatile state is purely for optimization and can be reconstructed after a restart without compromising safety.

Volatile state is maintained in memory and rebuilt on server restart. It includes:

1. **`commitIndex`** : Index of highest log entry known to be committed (initialized to 0, increases monotonically)
2. **`lastApplied`** : Index of highest log entry applied to state machine (initialized to 0, ≤ `commitIndex`)
3. **`nextIndex[]`** : For each server, index of the next log entry to send to that server (initialized to leader's last log index + 1)
4. **`matchIndex[]`** : For each server, index of highest log entry known to be replicated on server (initialized to 0)

These fields are reconstructed after a crash:

- `commitIndex` and `lastApplied` can be reset to 0 and rebuilt by replaying the persisted log
- `nextIndex` and `matchIndex` are leader-only state that can be reinitialized when a server becomes leader

The table below details each volatile field's role:

Field Name	Type	Description	Reconstruction Method After Crash
`commitIndex`	`int`	Index of highest committed entry. Leader updates based on quorum replication; followers update from leader's AppendEntries.	Initialize to 0, then scan persisted log for highest entry where a majority of servers have replicated it (requires communication with peers).
`lastApplied`	`int`	Index of highest applied entry to state machine. Incremented after applying an entry.	Initialize to 0, then replay all committed entries from log[1] to log[commitIndex] through state machine.
`nextIndex[]`	`int` slice	Leader-only: for each follower, index of next entry to send. Optimistic guess (leader last index + 1).	When becoming leader, initialize to `lastLogIndex + 1` for all followers.
`matchIndex[]`	`int` slice	Leader-only: for each follower, index of highest known replicated entry. Pessimistic (0).	When becoming leader, initialize to 0 for all followers, then update as followers acknowledge entries.

> **Important Distinction**: Note that `commitIndex` and `lastApplied` exist on *all* servers (followers learn `commitIndex` from the leader's AppendEntries RPC), while `nextIndex` and `matchIndex` exist *only* on the leader. This reflects Raft's asymmetry: followers are mostly passive, while the leader actively manages replication.

State Reconstruction Example: After a crash and restart, a Raft node:

1. Loads persistent state from disk (`currentTerm=5` , `votedFor=2` , `log` with 10 entries)
2. Initializes volatile state: `commitIndex=0` , `lastApplied=0` , `nextIndex=nil` , `matchIndex=nil`
3. Starts as a follower, receives AppendEntries from the current leader (term 5)
4. Learns the leader's `commitIndex=8` from the RPC, sets its own `commitIndex=8`
5. Applies entries 1 through 8 to its state machine, advancing `lastApplied` to 8
6. If later elected leader, initializes `nextIndex` to 11 (10+1) for each follower

ADR: Storing nextIndex/matchIndex as Slices vs. Maps

Context: We need to track replication progress for each peer. Peers are identified by integer IDs, but the set of peers may change during configuration changes.

Options Considered:

1. **Slice indexed by peer ID**: `nextIndex[peerID]` directly accessed
2. **Map from peer ID to value**: `nextIndex map[int]int`

Option	Pros	Cons
Slice by ID	- O(1) access, memory-efficient for dense IDs - Simple iteration with for-loop	- Wastes memory if IDs are sparse - Requires knowing max ID - Difficult configuration changes (need to resize)
Map	- Memory-efficient for sparse IDs - Easy to add/remove peers during configuration changes	- Slower access than array - More complex iteration - Go's map iteration order non-deterministic

Decision: Use slices indexed by peer ID, assuming dense integer IDs (0..N-1 for N servers).

Rationale: Most Raft implementations assume a small, fixed cluster size with sequential IDs. Slices provide better performance for the frequent operations in the replication loop (accessing `nextIndex[i]` for each heartbeat). Configuration changes (Milestone 4) will require careful handling, but the simplicity and performance justify this choice for the learning implementation.

Consequences: We must ensure peer IDs are contiguous integers starting from 0 or 1. Adding/removing servers requires reindexing these slices, which is acceptable for our educational context.

4.3 Messages and RPC Structures

Mental Model: Certified Mail with Return Receipt Imagine sending important legal documents via certified mail. The envelope must contain: (1) the current date (term), (2) a reference to the previous document you're building upon (prevLogIndex/Term), (3) the new documents themselves (entries), and (4) instructions about which documents are officially filed (leaderCommit). The return receipt confirms receipt and indicates whether the documents were accepted or rejected (with reasons). Raft's RPCs follow this exact pattern—every message carries enough context to detect stale information and ensure consistency.

Raft uses two key RPCs for its operation, each with carefully designed request and response structures:

RequestVote RPC: Used by candidates to gather votes during elections

RequestVote RPC Arguments (sent by candidate to all servers):

Field Name	Type	Description	Purpose in Protocol
`Term`	`int`	Candidate's term.	Ensures recipient can detect stale candidates (lower term) and update their own term if candidate's is higher.
`CandidateId`	`int`	Candidate requesting vote.	Identifies who is asking, used for recording `votedFor`.
`LastLogIndex`	`int`	Index of candidate's last log entry.	Used by voter to apply "up-to-date" check: candidate's log must be at least as current as voter's.
`LastLogTerm`	`int`	Term of candidate's last log entry.	Combined with `LastLogIndex` for accurate log comparison (terms more important than length).

RequestVote RPC Results (returned by recipient to candidate):

Field Name	Type	Description	Purpose in Protocol
`Term`	`int`	Current term of recipient, for candidate to update itself.	Allows candidate to detect if it's outdated (higher term received) and revert to follower.
`VoteGranted`	`bool`	True means candidate received vote.	Indicates whether the election tally should include this vote.

AppendEntries RPC: Used by leader to replicate log entries and as heartbeat

AppendEntries RPC Arguments (sent by leader to all followers, potentially as empty heartbeat):

Field Name	Type	Description	Purpose in Protocol
`Term`	`int`	Leader's term.	Allows followers to detect stale leaders and update their term.
`LeaderId`	`int`	So follower can redirect clients.	Not strictly necessary for protocol but useful for client redirection.
`PrevLogIndex`	`int`	Index of log entry immediately preceding new ones.	Consistency check: follower must have matching entry here, or reject.
`PrevLogTerm`	`int`	Term of `PrevLogIndex` entry.	Paired with `PrevLogIndex` for exact matching (not just index).
`Entries[]`	`[]LogEntry`	Log entries to store (empty for heartbeat).	The actual data to replicate; may be multiple entries for efficiency.
`LeaderCommit`	`int`	Leader's `commitIndex`.	Tells follower how far it can safely apply entries to its state machine.

AppendEntries RPC Results (returned by follower to leader):

Field Name	Type	Description	Purpose in Protocol
`Term`	`int`	Current term of follower, for leader to update itself.	Allows leader to detect if it's outdated and step down.
`Success`	`bool`	True if follower contained entry matching `PrevLogIndex` and `PrevLogTerm`.	If false, leader decrements `nextIndex` and retries with earlier entry.

> **Critical Detail**: The `PrevLogIndex` and `PrevLogTerm` fields implement Raft's **log matching property**. By requiring an exact match at the previous index, the follower can detect inconsistencies and force the leader to backtrack. This is the mechanism that ensures all logs eventually converge to the same state.

Message Format Unification: In our implementation, we'll wrap these RPCs in a general `Message` structure for transport abstraction:

Field Name	Type	Description	Purpose
`Type`	`string`	Message type: "RequestVote", "RequestVoteResponse", "AppendEntries", "AppendEntriesResponse".	Allows single handler to dispatch to appropriate logic.
`From`	`int`	Sender ID.	For debugging and potentially for redirecting clients.
`To`	`int`	Recipient ID.	Used by transport layer to route message.
`Payload`	`json.RawMessage`	JSON-encoded RPC arguments or results.	Flexible container for different message types without separate structs.

ADR: Unified Message Envelope vs. Separate RPC Structs

Context: We need to send different types of RPCs (RequestVote, AppendEntries) over a common transport layer. The transport needs to know basic routing information (to/from) and how to deserialize the content.

Options Considered:

1. **Unified envelope with type field and JSON payload**: Single `Message` struct with `Type` and `Payload` fields
2. **Separate structs per RPC type**: Distinct `RequestVoteMsg`, `AppendEntriesMsg`, each implementing a common interface
3. **Union type with discriminators**: Go interface with type switch, each RPC as separate concrete type

Option	Pros	Cons
Unified envelope	- Single handler function with switch on Type - Easy to add new message types - Simple serialization/deserialization	- Runtime type checking (string comparison) - JSON overhead for every message - Less type safety
Separate structs	- Compile-time type safety - Can use efficient binary encoding per type - Clear API boundaries	- Multiple handler functions or large type switch - More boilerplate code - Harder to extend
Union interface	- Go-idiomatic (interface + type switch) - Type-safe once deserialized - Can use different encodings per type	- Complex deserialization (need custom unmarshaler) - More complex code structure

Decision: Use unified `Message` envelope with JSON payload for the learning implementation.

Rationale: The primary goal is understandability and ease of debugging. JSON payloads are human-readable in logs, and a single message structure simplifies the transport layer. While not production-optimal (performance overhead, no compile-time checking), it's perfect for an educational implementation where clarity trumps efficiency.

Consequences: We'll need to carefully handle JSON serialization/deserialization and ensure the `Type` field is always set correctly. Debugging will be easier since we can log entire messages as JSON strings.

Example RPC Flow (RequestVote):

1. Candidate with ID=3, term=5, last log index=10, last log term=4 creates:

```json
{
  "Type": "RequestVote",
  "From": 3,
  "To": 1,
  "Payload": {
    "Term": 5,
    "CandidateId": 3,
    "LastLogIndex": 10,
    "LastLogTerm": 4
  }
}
```

2. Server 1 receives, deserializes, processes, responds with:

```json
{
  "Type": "RequestVoteResponse",
  "From": 1,
  "To": 3,
  "Payload": {
    "Term": 5,
    "VoteGranted": true
  }
}
```

Common Pitfalls in Data Model Design

⚠️ Pitfall: Treating Term as a Lamport Clock

- **Description**: Implementing term comparisons and increments like a Lamport timestamp (max(sender.term, receiver.term) + 1).
- **Why It's Wrong**: Terms are leadership epochs, not logical clocks. A follower receiving a higher term must adopt it exactly (not increment it further) and revert to follower state. Incrementing could create a term never used by any leader.
- **Fix**: Always set `currentTerm = max(currentTerm, receivedTerm)` without adding 1, unless starting an election.

⚠️ Pitfall: Zero vs One-Based Indexing Confusion

- **Description**: Inconsistently treating log indices as 0-based (Go slice style) vs 1-based (Raft paper style).
- **Why It's Wrong**: The Raft paper defines index 0 as a sentinel entry (term 0, no command). Off-by-one errors in `prevLogIndex` or `commitIndex` calculations break log consistency checks.
- **Fix**: Stick to Raft's 1-based indexing internally. Store entries in a slice starting at position 0, but treat `log[0]` as a dummy entry with term 0, index 0, and all real entries from index 1 at `log[1]`.

⚠️ Pitfall: Forgetting to Persist votedFor Before Responding

- **Description**: Updating `votedFor` in memory and granting a vote without first persisting to disk.
- **Why It's Wrong**: If the server crashes after granting vote but before persisting, it may reboot, forget it voted, and grant another vote in the same term, potentially enabling two leaders.
- **Fix**: Always write `votedFor` to stable storage (or our `MemoryStore`) before sending `VoteGranted=true`.

⚠️ Pitfall: Storing nextIndex/matchIndex for All Servers Instead of Just Followers

- **Description**: Including the leader's own ID in `nextIndex` and `matchIndex` slices.
- **Why It's Wrong**: The leader doesn't need to replicate entries to itself. Including it in quorum calculations (like `matchIndex` for commit advancement) incorrectly counts the leader twice.
- **Fix**: Only maintain `nextIndex` and `matchIndex` for follower peers, not including self. When checking for quorum, explicitly add the leader's own log.

Implementation Guidance

A. Technology Recommendations Table:

Component	Simple Option	Advanced Option
Persistent Storage	In-memory `MemoryStore` with periodic mock persistence	BoltDB, SQLite, or direct file I/O with checksums
Serialization	JSON via `encoding/json`	Protocol Buffers, Cap'n Proto, or custom binary format
Transport Layer	HTTP/1.1 with JSON payloads	gRPC with streaming, WebSockets, or raw TCP

B. Recommended File/Module Structure:

```
raft/
├── internal/
│   ├── raft/
│   │   ├── node.go           # RaftNode struct and main logic
│   │   ├── state.go          # Persistent/volatile state definitions
│   │   ├── messages.go       # RPC structs and Message wrapper
│   │   ├── logstore.go       # LogEntry and LogStore interface
│   │   └── types.go          # Constants (NodeState, etc.)
│   ├── storage/
│   │   ├── memory_store.go   # MemoryStore implementation
│   │   └── persistent.go     # Interface for disk persistence
│   └── transport/
│       ├── http.go           # HTTPServer implementation
│       └── interface.go      # Transport interface
├── cmd/
│   └── server/
│       └── main.go           # Entry point for each node
└── pkg/
    └── state_machine/
        └── interface.go      # StateMachine interface
```

C. Infrastructure Starter Code (Complete):

GO

```go
// raft/internal/raft/types.go

package raft


type NodeState int


const (

    Follower NodeState = iota

    Candidate

    Leader

)


// raft/internal/raft/messages.go

package raft


import "encoding/json"


type Message struct {

    Type    string          `json:"type"`

    From    int             `json:"from"`

    To      int             `json:"to"`

    Payload json.RawMessage `json:"payload"`

}


// RequestVote RPC structures

type RequestVoteArgs struct {

    Term         int `json:"term"`

    CandidateID  int `json:"candidateId"`

    LastLogIndex int `json:"lastLogIndex"`

    LastLogTerm  int `json:"lastLogTerm"`

}


type RequestVoteReply struct {

    Term        int  `json:"term"`

    VoteGranted bool `json:"voteGranted"`

}


// AppendEntries RPC structures

type AppendEntriesArgs struct {

    Term         int        `json:"term"`

    LeaderID     int        `json:"leaderId"`
```

```
    PrevLogIndex int        `json:"prevLogIndex"`
    PrevLogTerm  int        `json:"prevLogTerm"`
    Entries      []LogEntry `json:"entries"`
    LeaderCommit int        `json:"leaderCommit"`
}

type AppendEntriesReply struct {
    Term    int  `json:"term"`
    Success bool `json:"success"`
}

// raft/internal/raft/logstore.go
package raft

type LogEntry struct {
    Term    int         `json:"term"`
    Index   int         `json:"index"`   // Note: should equal position in slice + 1
    Command interface{} `json:"command"`
}

// LogStore interface abstracts log storage
type LogStore interface {
    Append(entries []LogEntry) error
    Get(index int) (LogEntry, error)
    LastIndex() int
    GetTerm(index int) (int, error)
    Truncate(fromIndex int) error
    // For persistence:
    GetCurrentTerm() int
    SetCurrentTerm(term int) error
    GetVotedFor() int
    SetVotedFor(candidateID int) error
    GetLogSlice(start, end int) ([]LogEntry, error)
}

// raft/internal/storage/memory_store.go
package storage

import (
```

```
    "sync"
    "raft/internal/raft"
)

type MemoryStore struct {
    mu          sync.RWMutex
    entries     []raft.LogEntry
    currentTerm int
    votedFor    int
}

func NewMemoryStore() *MemoryStore {
    // Create with dummy entry at index 0
    return &MemoryStore{
        entries: []raft.LogEntry{{Term: 0, Index: 0}},
        votedFor: -1,
    }
}

func (m *MemoryStore) Append(entries []raft.LogEntry) error {
    m.mu.Lock()
    defer m.mu.Unlock()

    for i, entry := range entries {
        // Ensure index matches position
        expectedIndex := len(m.entries)
        if entry.Index != expectedIndex {
            entry.Index = expectedIndex
        }
        m.entries = append(m.entries, entry)
    }
    return nil
}

func (m *MemoryStore) Get(index int) (raft.LogEntry, error) {
    m.mu.RLock()
    defer m.mu.RUnlock()
```

```
    if index < 0 || index >= len(m.entries) {
        return raft.LogEntry{}, fmt.Errorf("index out of bounds: %d", index)
    }
    return m.entries[index], nil
}

func (m *MemoryStore) LastIndex() int {
    m.mu.RLock()
    defer m.mu.RUnlock()
    return len(m.entries) - 1
}

func (m *MemoryStore) GetTerm(index int) (int, error) {
    if index == 0 {
        return 0, nil
    }
    entry, err := m.Get(index)
    if err != nil {
        return 0, err
    }
    return entry.Term, nil
}

func (m *MemoryStore) Truncate(fromIndex int) error {
    m.mu.Lock()
    defer m.mu.Unlock()

    if fromIndex <= 0 || fromIndex >= len(m.entries) {
        return fmt.Errorf("invalid truncation index: %d", fromIndex)
    }
    m.entries = m.entries[:fromIndex]
    return nil
}

func (m *MemoryStore) GetCurrentTerm() int {
    m.mu.RLock()
    defer m.mu.RUnlock()
    return m.currentTerm
```

```
}

func (m *MemoryStore) SetCurrentTerm(term int) error {
    m.mu.Lock()
    defer m.mu.Unlock()
    m.currentTerm = term
    return nil
}

func (m *MemoryStore) GetVotedFor() int {
    m.mu.RLock()
    defer m.mu.RUnlock()
    return m.votedFor
}

func (m *MemoryStore) SetVotedFor(candidateID int) error {
    m.mu.Lock()
    defer m.mu.Unlock()
    m.votedFor = candidateID
    return nil
}

func (m *MemoryStore) GetLogSlice(start, end int) ([]raft.LogEntry, error) {
    m.mu.RLock()
    defer m.mu.RUnlock()

    if start < 0 || end > len(m.entries) || start > end {
        return nil, fmt.Errorf("invalid range: %d-%d", start, end)
    }
    // Return copy to avoid race conditions
    result := make([]raft.LogEntry, end-start)
    copy(result, m.entries[start:end])
    return result, nil
}
```

D. Core Logic Skeleton Code (TODOs):

```go
// raft/internal/raft/state.go

package raft


// PersistentState holds the state that must survive crashes

type PersistentState struct {

    // TODO 1: Define fields currentTerm, votedFor, and log reference

    // TODO 2: Add methods to safely access/modify these fields

    // TODO 3: Ensure all modifications go through the LogStore interface

}


// VolatileState holds in-memory state that can be reconstructed

type VolatileState struct {

    // TODO 4: Define commitIndex, lastApplied, nextIndex, matchIndex

    // TODO 5: Add initialization logic for when node becomes leader

    // TODO 6: Add method to update commitIndex based on matchIndex quorum

}


// raft/internal/raft/node.go (partial)

package raft


type RaftNode struct {

    // TODO 7: Define all fields from naming conventions

    // TODO 8: Initialize persistent and volatile states in NewRaftNode

    // TODO 9: Implement run() method that calls appropriate state handler

}


func NewRaftNode(id int, peers []int, sm StateMachine, transport Transport) *RaftNode {

    // TODO 10: Create and initialize RaftNode

    // TODO 11: Set initial state to Follower

    // TODO 12: Initialize election timeout with random duration

    // TODO 13: Start background goroutine for run() method

}


func (rn *RaftNode) handleRequestVote(args RequestVoteArgs) RequestVoteReply {

    // TODO 14: Implement RequestVote RPC handler

    // TODO 15: Check if candidate's term is at least currentTerm

    // TODO 16: Check votedFor: either nil or same candidate in current term

    // TODO 17: Check if candidate's log is at least as up-to-date as local log

    // TODO 18: If all checks pass, persist votedFor before returning
```

```
    // TODO 19: Always update currentTerm if candidate's term is higher
}

func (rn *RaftNode) handleAppendEntries(args AppendEntriesArgs) AppendEntriesReply {
    // TODO 20: Implement AppendEntries RPC handler
    // TODO 21: Reject if leader's term < currentTerm
    // TODO 22: Reset election timeout on valid leader communication
    // TODO 23: Check log consistency at PrevLogIndex
    // TODO 24: If mismatch, return false so leader can backtrack
    // TODO 25: If match, append new entries (truncating if conflict)
    // TODO 26: Update commitIndex if LeaderCommit > current commitIndex
}
```

E. Language-Specific Hints (Go):

- Use `sync.RWMutex` for protecting shared state in `MemoryStore` and `RaftNode`
- Use `time.Timer` for election timeout (not `time.Sleep`) to allow cancellation
- For JSON serialization of `interface{}` commands, consider using `json.RawMessage` to defer parsing
- Implement `String()` methods on `NodeState` and other enums for readable logging
- Use 1-based indexing consistently: `log[0]` is dummy entry, real entries start at `log[1]`

F. Milestone Checkpoint for Data Model: After implementing the data structures but before implementing the full protocol, you should be able to:

1. Create a `MemoryStore` and verify it stores/retrieves log entries correctly:

```bash
go test ./internal/storage/... -v -run TestMemoryStore
```

2. Create a `RaftNode` with mock transport and verify initialization:

```bash
go test ./internal/raft/... -v -run TestNewRaftNode
```

3. Serialize and deserialize RPC messages:

```bash
go test ./internal/raft/... -v -run TestMessageSerialization
```

Expected output should show:

- MemoryStore correctly maintains log indices
- RaftNode initializes with term=0, state=Follower
- Messages can be round-trip serialized through JSON

Common issues to check:

- Log index off-by-one errors (are you treating index 0 as dummy?)
- Mutex not held during state updates (data races)
- JSON field tags missing causing serialization failures

G. Debugging Tips for Data Model Issues:

Symptom	Likely Cause	How to Diagnose	Fix
"Index out of bounds" when accessing log[0]	Not initializing with dummy entry at index 0	Check MemoryStore constructor	Add dummy entry with Term=0, Index=0
"Term comparison fails" after restart	Not persisting/loading currentTerm	Add logging before/after persistence	Ensure SetCurrentTerm writes to stable storage
"Vote granted twice in same term"	Not persisting votedFor before responding	Add debug log when granting vote	Call SetVotedFor before returning VoteGranted=true
"JSON unmarshal error" on RPCs	Field names mismatch between struct and JSON	Print raw JSON and compare tags	Ensure struct fields have correct json:"fieldName" tags
"commitIndex > lastIndex"	Not bounds-checking commitIndex updates	Log commitIndex and lastIndex in AppendEntries	Add `min(leaderCommit, lastIndex)` logic

---

5. Component Design: Leader Election (Milestone 1)

> **Milestone(s):** 1 (Leader Election)

The leader election mechanism is the heartbeat of the Raft protocol, responsible for establishing a single authoritative node (the **leader**) to coordinate all operations. Without reliable leader election, the system cannot make progress, as followers won't know whose commands to follow. This section details the architectural decisions, state transitions, and safeguards that ensure exactly one leader emerges per **term**, even in the face of network delays and node failures.

5.1 Mental Model: The Student Council Election

Imagine a student council election where multiple candidates campaign for president. The election follows strict rules:

1. **Terms as Academic Years**: Each election happens in a specific academic year (term). If a candidate says they're running for "2024 term," but you hear the election has already moved to "2025 term," you ignore their outdated campaign.
2. **Campaign Timeouts**: Students patiently wait for announcements from the current president (leader). If they go too long without hearing any news, they assume the presidency is vacant and nominate themselves as candidates.
3. **Randomized Announcement Timing**: To prevent everyone from shouting "I'm running!" simultaneously and creating chaos, each student waits a *random* amount of time before announcing their candidacy. This gives one candidate a head start to gather votes.
4. **Voter Qualification Check**: Before voting, students check the candidate's qualifications (log completeness). They ask: "Have you attended at least as many important meetings (logged entries) as I have, and were you present for the most recent critical meeting (latest term)?" Only qualified candidates get votes.
5. **Majority Rule**: A candidate needs votes from a majority of the entire student body to win. If no one achieves a majority, the term ends without a president, and everyone waits for new random timeouts to try again in the next term.

This analogy maps directly to Raft:

- **Students** = Raft nodes
- **President** = Leader
- **Academic Year** = Term (monotonically increasing)
- **Announcements from President** = Heartbeat messages (`AppendEntries` RPCs)
- **Campaign Announcement** = `RequestVote` RPC
- **Meeting Attendance Record** = Log entries
- **Majority of Student Body** = Quorum (⌊n/2⌋ + 1)

The key insight is that this process ensures **safety** (no two presidents in the same year) while striving for **liveness** (eventually electing someone). The randomized timeouts and term-based logic resolve conflicts automatically.

5.2 Architecture Decision Records for Election

Decision: Randomized Election Timeouts vs. Fixed Timeouts

- **Context**: Nodes need to detect leader failure. A fixed timeout (e.g., 150ms for all nodes) would cause simultaneous candidacy announcements, leading to split votes where no candidate achieves a majority, repeatedly stalling the system.
- **Options Considered**:
  1. **Fixed Timeout**: All nodes use identical election timeout durations.
  2. **Randomized Timeout**: Each node picks a timeout uniformly from a range `[base, base+range]` after each reset.
  3. **Adaptive Timeout**: Dynamically adjust timeout based on network latency or past election history.
- **Decision**: Randomized timeout within a configured range (e.g., 150-300ms).
- **Rationale**:
  - **Simplicity**: Easy to implement with minimal state.
  - **Probabilistic Guarantee**: Given N nodes, the probability that *all* pick the exact same timeout is negligible. The first to timeout becomes candidate and has head start to gather votes.
  - **Deterministic Resolution**: Even if split votes occur (multiple candidates get votes but no majority), the next election round will have new random timeouts, eventually breaking symmetry.
  - **Stability**: The range is large enough to prevent collisions but small enough to ensure timely failure detection.
- **Consequences**:
  - **Enables**: Reliable leader election despite simultaneous failure detection.
  - **Trade-offs**: Slightly variable election duration; requires careful tuning of timeout range for specific deployment environments.

Option	Pros	Cons	Chosen?
Fixed Timeout	Predictable election timing	High probability of split votes (multiple candidates) leading to livelock	No
Randomized Timeout	Low probability of split votes; simple implementation	Election completion time varies	**Yes**
Adaptive Timeout	Could optimize for network conditions	Complex to implement; requires monitoring and tuning; risk of instability	No

Decision: Granting Votes Based on Log Completeness

- **Context**: A candidate with an outdated log must not become leader, as it could overwrite committed entries, violating **Leader Completeness** (safety property). The voting process must ensure the elected leader's log contains all committed entries.
- **Options Considered**:
  1. **First-Come-First-Served**: Grant vote to the first candidate that asks in the current term.
  2. **Term-Only Check**: Grant vote if candidate's term is ≥ current term.
  3. **Log Comparison**: Grant vote only if candidate's log is at least as up-to-date as own log (Raft paper §5.4.1).
- **Decision**: Use log comparison to determine "up-to-dateness."
- **Rationale**:
  - **Safety Guarantee**: Ensures the **Leader Completeness Property**—any leader elected for a given term contains all entries committed in previous terms.
  - **Formal Definition**: Candidate's log is "more up-to-date" if: a) The last entry in the logs have different terms: candidate with higher term wins. b) The last entry terms are equal: candidate with longer log (higher index) wins.
  - **Implementation Simplicity**: Simple comparison of `(lastLogTerm, lastLogIndex)` tuples.
- **Consequences**:
  - **Enables**: Safety property that committed entries survive leader changes.
  - **Trade-offs**: A node with a more complete log might not win if it's slow to start election (liveness vs. safety trade-off resolved by randomized timeouts).

Option	Pros	Cons	Chosen?
First-Come-First-Served	Simple; fast election	Elects outdated leaders, violating safety	No
Term-Only Check	Simple; prevents old-term candidates	Still elects leaders with missing committed entries	No
Log Comparison	Ensures leader completeness safety property	Slightly more complex comparison logic	**Yes**

Decision: Handling Split Votes with Term Incrementation

- **Context**: When votes split (no candidate achieves majority), the election fails for that term. The system must progress rather than stall indefinitely.
- **Options Considered**:
  1. **Same Term Retry**: Candidates retry election immediately in the same term with new randomized timeouts.
  2. **New Term Retry**: All nodes increment term and start new election (Raft paper approach).
  3. **Backoff and Retry**: Exponential backoff before retrying in same term.
- **Decision**: Increment term and start new election when election timeout expires without victory.
- **Rationale**:
  - **Progress Guarantee**: Increasing term ensures forward progress—each failed election moves to a higher term where random timeouts are re-randomized, breaking symmetry.
  - **Simplicity**: Clean state transition: candidate → follower (on new term) → candidate again.
  - **Prevents Stale Leaders**: A leader from previous term cannot disrupt current election because its term is outdated.
  - **Aligns with RPC Semantics**: All RPCs include term; higher term forces receivers to update their term and revert to follower.
- **Consequences**:
  - **Enables**: Eventual leader election despite repeated split votes.
  - **Trade-offs**: Rapid term growth during unstable network; requires careful term comparison logic in all RPC handlers.

Option	Pros	Cons	Chosen?
Same Term Retry	Maintains lower term numbers	High probability of repeated split votes (livelock)	No
New Term Retry	Breaks symmetry via new random timeouts; ensures progress	Terms increase quickly during network issues	**Yes**
Backoff and Retry	Reduces election frequency	Complex backoff logic; still risks livelock in same term	No

5.3 Common Pitfalls in Leader Election

⚠️ Pitfall: Forgetting to Reset Election Timer on Valid Leader Communication

- **Description**: A follower receives an `AppendEntries` RPC from a valid leader (term ≥ currentTerm) but fails to reset its election timeout timer. This causes unnecessary elections even when the leader is healthy.
- **Why It's Wrong**: Violates liveness—the system constantly elects new leaders, causing instability and preventing log replication from progressing.
- **How to Fix**: Every time a follower receives an `AppendEntries` RPC with `term >= currentTerm`, reset the election timer. This includes both heartbeat messages (empty entries) and actual log replication messages.

⚠️ Pitfall: Incorrect Term Comparison in RequestVote Handler

- **Description**: The `handleRequestVote` logic compares candidate's term with currentTerm but uses wrong operators (e.g., `>` instead of `>=`), causing nodes to reject valid candidates or accept outdated ones.
- **Why It's Wrong**: May prevent a legitimate leader election or allow a candidate from an older term to receive votes, violating **Election Safety** (two leaders in same term).
- **How to Fix**: Follow the Raft paper precisely:
  1. If `args.Term < currentTerm`: reply false.
  2. If `args.Term > currentTerm`: update `currentTerm = args.Term`, transition to follower, and then continue voting logic.
  3. Only grant vote if `votedFor` is null or equals `candidateId`, **and** candidate's log is at least as up-to-date.

⚠️ Pitfall: Not Persisting votedFor Before Sending Vote Grant

- **Description**: A node updates its `votedFor` in memory and sends a `RequestVoteReply` granting the vote, but crashes before persisting `votedFor` to stable storage. After restart, it forgets it voted and may vote for another candidate in the same term.
- **Why It's Wrong**: Could lead to double voting in the same term, potentially enabling two candidates to achieve majority, violating **Election Safety**.
- **How to Fix**: **Always persist state changes before taking action that depends on that state**. Before sending `VoteGranted = true`, write `(currentTerm, votedFor)` to stable storage (e.g., disk). Use write-ahead logging pattern.

⚠️ Pitfall: Mishandling the votedFor Field Across Term Boundaries

- **Description**: A node persists `votedFor` for term 3, then later increments its `currentTerm` to 4 but forgets to clear `votedFor` (set to null). When a candidate from term 4 requests a vote, the node incorrectly rejects because `votedFor` still points to a candidate from term 3.
- **Why It's Wrong**: Prevents legitimate elections in new terms, harming liveness.
- **How to Fix**: When incrementing `currentTerm` (e.g., on discovering a higher term or starting a new election), **always** set `votedFor = null` and persist both fields together atomically.

⚠️ Pitfall: Not Randomizing Election Timeout Per Reset

- **Description**: A node picks a random election timeout once at startup and reuses the same value every time the timer resets. If multiple nodes happen to pick similar values, split votes persist across election rounds.
- **Why It's Wrong**: Reduces the effectiveness of randomization; split vote scenarios may not resolve.
- **How to Fix**: Generate a new random timeout within the configured range **each time** `resetElectionTimeout()` is called (e.g., on receiving valid leader RPC or when transitioning to candidate).

5.4 Implementation Guidance for Milestone 1

This section provides concrete starting points for implementing leader election in Go. We separate infrastructure code (provided) from core logic (you implement via TODOs).

A. Technology Recommendations Table

Component	Simple Option (Recommended)	Advanced Option
Timer Infrastructure	`time.Ticker` + `select` in goroutine	Custom timer wheel for scalability
Random Number Generation	`math/rand` with seeding per node	Cryptographic randomness for security
State Persistence	In-memory map (for testing) → File + `encoding/gob`	BadgerDB/LevelDB for production
RPC Transport	HTTP + JSON (`net/http` , `encoding/json`)	gRPC + Protobuf for performance

B. Recommended File/Module Structure

For Milestone 1, focus on these files:

```
project-root/
├── cmd/
│   └── server/               # Entry points for each node
│       └── main.go
├── internal/
│   ├── raft/                 # Core Raft implementation
│   │   ├── node.go           # RaftNode struct and main loop (TODOs here)
│   │   ├── state.go          # PersistentState, VolatileState, LogEntry
│   │   ├── election.go       # Candidate logic, RequestVote handler (TODOs)
│   │   ├── timer.go          # Timer infrastructure (provided)
│   │   └── transport.go      # Abstract transport interface
│   ├── store/
│   │   └── memory.go         # MemoryStore (provided)
│   └── transport/
│       └── http.go           # HTTPServer implementation (provided)
└── go.mod
```

C. Infrastructure Starter Code

Timer Infrastructure (Complete) – Copy this into `internal/raft/timer.go`:

```go
package raft

import (
    "math/rand"
    "sync"
    "time"
)

// Timer manages election timeouts with random durations.
type Timer struct {
    mu            sync.Mutex
    duration      time.Duration
    timer         *time.Timer
    resetChan     chan struct{}
    timeoutChan   chan struct{}
    stopped       bool
    baseTimeout   time.Duration
    rangeTimeout  time.Duration
}

// NewTimer creates a new election timer with random range [base, base+range].
func NewTimer(base, rangeDur time.Duration) *Timer {
    t := &Timer{
        baseTimeout:  base,
        rangeTimeout: rangeDur,
        resetChan:    make(chan struct{}, 1),
        timeoutChan:  make(chan struct{}, 1),
    }
    t.reset()
    go t.run()
    return t
}

// reset generates a new random timeout and resets the internal timer.
func (t *Timer) reset() {
    t.mu.Lock()
    defer t.mu.Unlock()
```

```
    // Generate random duration within range
    randRange := time.Duration(rand.Int63n(int64(t.rangeTimeout)))
    t.duration = t.baseTimeout + randRange

    if t.timer != nil {
        t.timer.Stop()
    }
    t.timer = time.NewTimer(t.duration)
    t.stopped = false
}

// Run monitors the timer and sends to timeoutChan on expiration.
func (t *Timer) run() {
    for {
        select {
        case <-t.timer.C:
            t.mu.Lock()
            if !t.stopped {
                select {
                case t.timeoutChan <- struct{}{}:
                default:
                }
            }
            t.mu.Unlock()
        case <-t.resetChan:
            t.reset()
        }
    }
}

// Reset signals the timer to generate a new random timeout and restart.
func (t *Timer) Reset() {
    select {
    case t.resetChan <- struct{}{}:
    default:
    }
}
```

```
// TimeoutChan returns the channel that receives events when timer expires.
func (t *Timer) TimeoutChan() <-chan struct{} {
    return t.timeoutChan
}

// Stop halts the timer (for node shutdown).
func (t *Timer) Stop() {
    t.mu.Lock()
    defer t.mu.Unlock()
    if t.timer != nil {
        t.timer.Stop()
    }
    t.stopped = true
}
```

Memory Store (Complete) – Copy this into `internal/store/memory.go`:

```go
package store

import (
    "sync"
    "github.com/yourusername/raft/internal/raft"
)

// MemoryStore implements a volatile in-memory log store for testing.
type MemoryStore struct {
    mu        sync.RWMutex
    entries   []raft.LogEntry
    currentTerm int
    votedFor    int
}

// NewMemoryStore creates an empty memory store.
func NewMemoryStore() *MemoryStore {
    return &MemoryStore{
        entries: make([]raft.LogEntry, 0),
        // Index 0 is dummy entry
        entries = append(entries, raft.LogEntry{Term: 0, Index: 0})
    }
}

func (s *MemoryStore) Append(entries []raft.LogEntry) error {
    s.mu.Lock()
    defer s.mu.Unlock()
    s.entries = append(s.entries, entries...)
    return nil
}

func (s *MemoryStore) Get(index int) (raft.LogEntry, error) {
    s.mu.RLock()
    defer s.mu.RUnlock()
    if index < 0 || index >= len(s.entries) {
        return raft.LogEntry{}, errors.New("index out of bounds")
    }
    return s.entries[index], nil
```

```
}

func (s *MemoryStore) LastIndex() int {
    s.mu.RLock()
    defer s.mu.RUnlock()
    return len(s.entries) - 1
}

func (s *MemoryStore) GetTerm(index int) (int, error) {
    if index == 0 {
        return 0, nil
    }
    entry, err := s.Get(index)
    if err != nil {
        return 0, err
    }
    return entry.Term, nil
}

func (s *MemoryStore) Truncate(fromIndex int) error {
    s.mu.Lock()
    defer s.mu.Unlock()
    if fromIndex <= 0 || fromIndex >= len(s.entries) {
        return errors.New("invalid truncate index")
    }
    s.entries = s.entries[:fromIndex]
    return nil
}

func (s *MemoryStore) GetPersistentState() (int, int) {
    s.mu.RLock()
    defer s.mu.RUnlock()
    return s.currentTerm, s.votedFor
}

func (s *MemoryStore) SetPersistentState(term, votedFor int) {
    s.mu.Lock()
    defer s.mu.Unlock()
```

```
    s.currentTerm = term

    s.votedFor = votedFor

}
```

D. Core Logic Skeleton Code

Main Node Loop (TODOs) – Add to `internal/raft/node.go`:

GO

```go
// run is the main event loop for the Raft node.
func (n *RaftNode) run() {
    for {
        switch n.state {
        case Follower:
            n.runFollower()
        case Candidate:
            n.runCandidate()
        case Leader:
            n.runLeader()
        }
    }
}

// runFollower handles follower state: waits for heartbeats or election timeout.
func (n *RaftNode) runFollower() {
    n.resetElectionTimeout()

    for n.state == Follower {
        select {
        case <-n.electionTimeoutTimer.TimeoutChan():
            // TODO 1: Election timeout expired
            // - Transition to Candidate state
            // - Increment currentTerm
            // - Vote for self
            // - Reset election timer with new random timeout
            // - Send RequestVote RPCs to all peers

        case rpc := <-n.transport.IncomingRPC():
            // TODO 2: Handle incoming RPCs
            // - If RPC term > currentTerm: update term and revert to follower
            // - Process RequestVote or AppendEntries based on RPC type
            // - If valid leader heartbeat (AppendEntries): reset election timer
        }
    }
}
```

```go
// resetElectionTimeout resets the election timer with a new random timeout.
func (n *RaftNode) resetElectionTimeout() {
    // TODO 3: Implement reset logic
    // - Call n.electionTimeoutTimer.Reset()
    // - This should generate a new random timeout within configured range
}

// runCandidate handles candidate state: starts election, collects votes.
func (n *RaftNode) runCandidate() {
    // TODO 4: Implement candidate logic
    // Step 1: Increment currentTerm
    // Step 2: Vote for self (set votedFor = self id)
    // Step 3: Persist (currentTerm, votedFor) to stable storage
    // Step 4: Reset election timer
    // Step 5: Send RequestVote RPCs to all peers in parallel

    votes := 1 // self-vote
    voteChan := make(chan bool, len(n.peers))

    for _, peer := range n.peers {
        go func(peer int) {
            args := RequestVoteArgs{
                Term:         n.currentTerm,
                CandidateID:  n.id,
                LastLogIndex: n.log.LastIndex(),
                LastLogTerm:  n.log.GetTerm(n.log.LastIndex()),
            }
            reply, err := n.transport.Send(peer, args)
            // TODO 5: Handle RPC response
            // - If reply.Term > currentTerm: revert to follower
            // - If vote granted: increment votes count
            // - If votes reach majority: transition to Leader
        }(peer)
    }

    for n.state == Candidate {
```

```
        select {
        case <-n.electionTimeoutTimer.TimeoutChan():
            // TODO 6: Election timeout - start new election
            // - Transition back to Candidate (which will increment term again)

        case voteGranted := <-voteChan:
            if voteGranted {
                votes++
                // TODO 7: Check if reached majority (len(peers)/2 + 1)
                // - If yes: transition to Leader
            }

        case rpc := <-n.transport.IncomingRPC():
            // TODO 8: Handle incoming RPCs while candidate
            // - If RPC term > currentTerm: update term, revert to follower
            // - If AppendEntries from valid leader: revert to follower
            // - If RequestVote with higher term: update term, revert to follower
        }
    }
}
```

RequestVote RPC Handler (TODOs) – Add to `internal/raft/election.go`:

```go
// handleRequestVote processes incoming vote requests.
func (n *RaftNode) handleRequestVote(args RequestVoteArgs) RequestVoteReply {
    n.mu.Lock()
    defer n.mu.Unlock()

    reply := RequestVoteReply{Term: n.currentTerm, VoteGranted: false}

    // TODO 9: Implement vote granting logic
    // Step 1: If args.Term < currentTerm: return false (reply.VoteGranted = false)

    // Step 2: If args.Term > currentTerm:
    // - Update currentTerm = args.Term
    // - Transition to Follower state
    // - Set votedFor = null
    // - Persist state

    // Step 3: Check if already voted this term
    // If votedFor is null or equals args.CandidateID, continue check

    // Step 4: Check log up-to-dateness
    // Candidate's log is more up-to-date if:
    // a) args.LastLogTerm > lastLogTerm, OR
    // b) args.LastLogTerm == lastLogTerm AND args.LastLogIndex >= lastLogIndex
    // where lastLogIndex = n.log.LastIndex(), lastLogTerm = n.log.GetTerm(lastLogIndex)

    // Step 5: If log check passes:
    // - Set votedFor = args.CandidateID
    // - Persist state (currentTerm, votedFor)
    // - Reset election timer
    // - Set reply.VoteGranted = true

    return reply
}
```

E. Language-Specific Hints

1. **Concurrency**: Use `sync.RWMutex` for protecting `RaftNode` fields. Acquire lock before accessing/modifying `currentTerm`, `votedFor`, `log`, or `state`.

2. **Time**: Use `time.Duration` for timeouts. Configure base timeout (e.g., 150ms) and range (e.g., 150ms) for 150-300ms random timeouts.
3. **Randomness**: Seed random generator once per node with unique seed (e.g., `rand.Seed(time.Now().UnixNano() + int64(nodeID))`).
4. **Goroutines**: Launch separate goroutines for each peer RPC call but collect responses via channels. Always handle goroutine leaks on node shutdown.
5. **JSON Serialization**: Use struct tags for RPC args/reply: `` `json:"term"` ``. Ensure all exported fields are serializable.

F. Milestone Checkpoint

After implementing Milestone 1, verify your leader election works:

1. **Start a 3-node cluster**:

```bash
go run cmd/server/main.go --id 1 --peers 2,3 --port 8080

go run cmd/server/main.go --id 2 --peers 1,3 --port 8081

go run cmd/server/main.go --id 3 --peers 1,2 --port 8082
```

2. **Expected Behavior**:
   - One node should print "Transitioning to Leader for term X"
   - Other nodes should print "Received vote request from node Y, granted: true/false"
   - No two nodes should claim leadership for the same term.
3. **Test Scenario - Kill Leader**:
   - Stop the leader process (Ctrl+C)
   - Within 150-300ms, one of the remaining nodes should time out and become candidate
   - It should win election and print leadership claim
4. **Test Scenario - Split Votes**:
   - Stop all nodes, restart simultaneously
   - Watch logs: multiple candidates may appear, but eventually one should win
   - Terms should increase with each failed election round
5. **Verification Command**:

```bash
# Run the provided test suite for election

go test ./internal/raft -run TestLeaderElection -v
```

Expected output: All tests pass, showing election safety and eventual liveness.

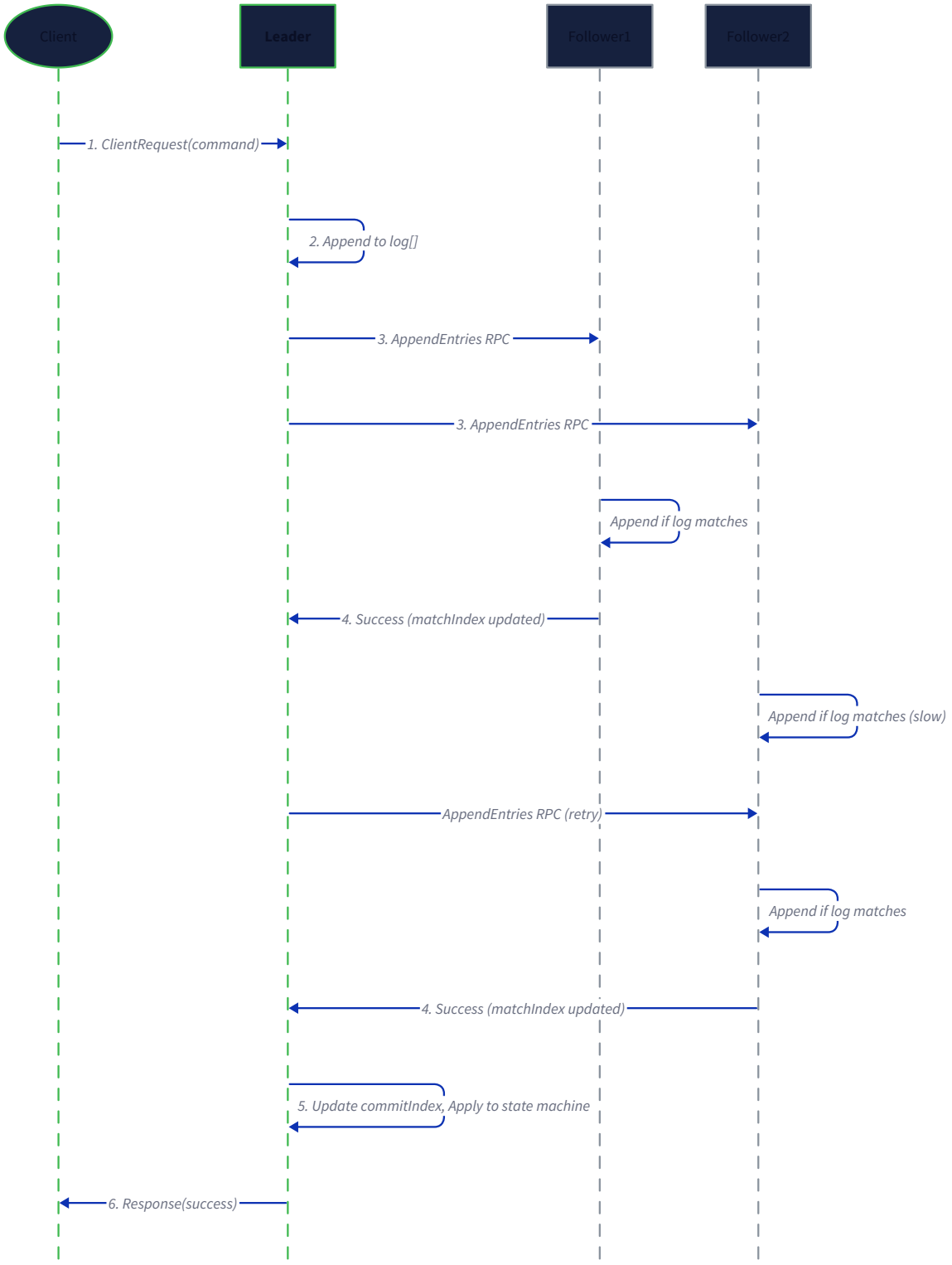
6. Component Design: Log Replication (Milestone 2)

> **Milestone(s):** 2 (Log Replication)

Log replication is the core mechanism by which Raft transforms individual server state machines into a **Replicated State Machine**. After leader election establishes an authoritative coordinator, replication ensures that all non-failed servers apply the same sequence of commands in the same order. This component implements the protocol's primary function: reliably copying log entries from the leader to followers while maintaining the **Log Matching Property** that guarantees consistency even during crashes and network partitions.

6.1 Mental Model: The Assembly Line with Quality Control

Imagine a manufacturing assembly line with a **foreman** (leader), several **assembly workers** (followers), and a **quality control inspector** (commit mechanism). The foreman receives customer orders (client commands) and writes them on individual work tickets (log entries). Each ticket includes a unique sequence number (index) and the foreman's current shift number (term).



Process Flow:

1. **Ticket Issuance**: The foreman writes a new ticket with the next available sequence number and attaches it to the master work board (leader's log).

2. **Work Distribution**: The foreman makes photocopies of the ticket and hands them to each assembly worker. With each ticket, the foreman specifies: "This is ticket #15, and it follows ticket #14 which was created during shift 5." This is the `PrevLogIndex` and `PrevLogTerm` in Raft's `AppendEntries` RPC.
3. **Quality Checkpoint**: Before accepting the new ticket, each worker checks their personal board: "Do I have ticket #14 from shift 5?" If yes, they staple the new ticket after it and signal acceptance. If not, they reject the ticket and request the foreman to provide missing tickets. This ensures **consistency** — workers only extend logs that match the leader's prefix.
4. **Production Sign-off**: Once the foreman receives confirmations from a **majority** of workers (including themselves), they mark the ticket as "approved for production" (committed). This quorum-based approval ensures that even if the foreman is replaced, the work won't be lost.
5. **Assembly Execution**: Each worker, upon seeing the approval mark advance, executes the instructions on all approved tickets in sequential order, building the final product (applying to state machine).

This assembly line analogy illustrates three critical aspects:

- **Log consistency checking** through the prevLogIndex/prevLogTerm mechanism
- **Quorum-based commitment** ensuring durability despite individual failures
- **Ordered execution** maintaining identical state across all workers

6.2 Architecture Decision Records for Replication

Decision: AppendEntries as Dual-Purpose RPC (Heartbeat + Replication)

> **Context**: The Raft protocol requires both leader heartbeats (to prevent unnecessary elections) and actual log entry replication. Implementing these as separate RPCs would double network traffic and complicate synchronization.

> **Options Considered**:
>
> 1. **Separate Heartbeat and AppendEntries RPCs**: Heartbeat messages without log entries sent periodically; AppendEntries only when new entries exist.
> 2. **Piggybacked Heartbeats**: Empty AppendEntries RPCs serve as heartbeats; entries included only when available.
> 3. **Dedicated Control Channel**: Continuous connection with multiplexed control messages.

> **Decision**: Use piggybacked heartbeats where empty `AppendEntries` RPCs serve as heartbeats, and entries are included when available.

> **Rationale**: The Raft paper's design unifies both functions: the same `AppendEntries` RPC structure handles empty entries (heartbeats) and non-empty entries (replication). This reduces protocol complexity, ensures heartbeats also verify log consistency, and minimizes code paths. The `Entries` slice being empty distinguishes a heartbeat from actual replication.

> **Consequences**:
>
> - **Positive**: Simpler implementation with fewer message types; heartbeats automatically repair minor log inconsistencies.
> - **Negative**: Slightly larger message size for heartbeats (still minimal); requires careful handling of empty vs. non-empty cases.
> - **Learning Benefit**: Demonstrates protocol optimization through message dual-purposing.

Option Comparison Table:

Option	Pros	Cons	Chosen?
Separate RPCs	Clear separation of concerns; potentially smaller heartbeat messages	Double network traffic; more complex state management; missed heartbeat might cause unnecessary election	❌
Piggybacked Heartbeats	Unified protocol; automatic log consistency checking; minimal additional complexity	Slightly larger heartbeat messages; must handle empty entries case	✅
Dedicated Control Channel	Real-time feedback; potential performance benefits	Significant implementation complexity; connection management overhead	❌

Decision: Log Matching Property Enforcement via PrevLogIndex/Term

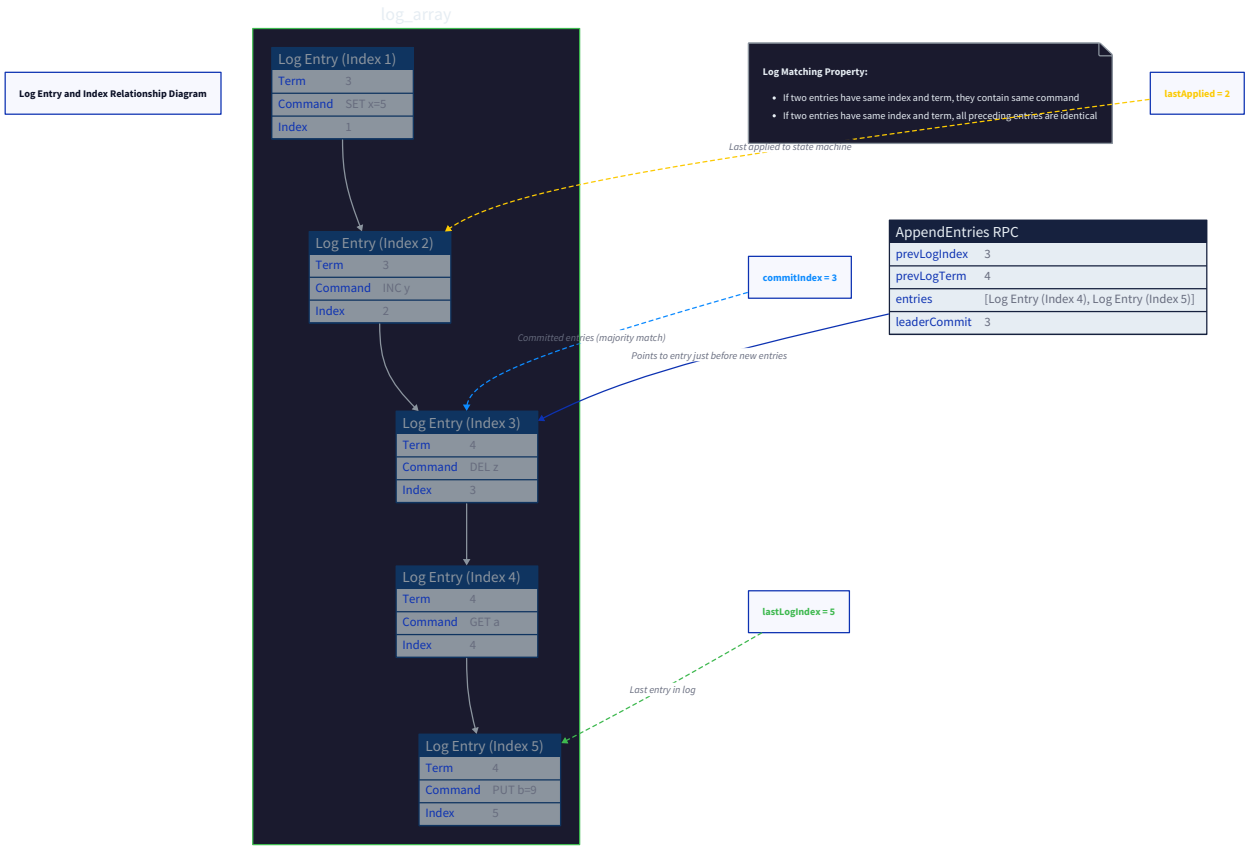
> **Context**: Followers must ensure their logs match the leader's log prefix before appending new entries. Without this check, divergent logs could create inconsistencies that violate state machine safety.

> **Options Considered**:
>
> 1. **Prefix Match Checking**: Leader sends previous entry's index and term; follower verifies match before appending.
> 2. **Checksum-based Verification**: Compute checksum of entire log prefix; send with entries.
> 3. **Append-Only with Rollback**: Append optimistically, detect mismatch later, then rollback.

> **Decision**: Implement prefix match checking using `PrevLogIndex` and `PrevLogTerm` as specified in the Raft paper.

> **Rationale**: The `PrevLogIndex` / `PrevLogTerm` mechanism provides a simple, efficient way to verify log consistency. If the follower's log doesn't contain an entry at `PrevLogIndex` with term `PrevLogTerm`, it rejects the `AppendEntries` request. This triggers the leader's **log backtracking** algorithm (decrementing `nextIndex` and retrying) until a matching point is found, at which point the follower truncates its divergent suffix and accepts the leader's entries. This approach guarantees the **Log Matching Property**: if two logs contain an entry with the same index and term, they are identical up to that index.

> **Consequences**:
>
> - **Positive**: Ensures strong consistency; enables efficient log repair; directly implements the Raft paper's specification.
> - **Negative**: May cause multiple round trips during log reconciliation; requires careful index management.
> - **Learning Benefit**: Teaches fundamental distributed systems concept of consensus through log consistency.



Log Consistency Check Algorithm:

1. **Leader Preparation**: Before sending entries starting at index `i`, leader includes `PrevLogIndex = i-1` and `PrevLogTerm = term at log[i-1]`.
2. **Follower Verification**: Follower checks if `log[PrevLogIndex].Term == PrevLogTerm`.
3. **Success Path**: If match exists, follower:
   - Truncates log from `PrevLogIndex+1` onward (if any entries exist)
   - Appends new entries
   - Updates `commitIndex` to `min(LeaderCommit, last new entry index)`
4. **Failure Path**: If no match, follower returns `Success=false`, causing leader to decrement `nextIndex` and retry.

Decision: Commit Index Advancement via Majority Replication

> **Context**: The leader must determine when a log entry is **committed** (durable and guaranteed to survive leader changes). Premature commitment risks data loss if a new leader doesn't have the entry; delayed commitment hurts availability.

> **Options Considered**:
>
> 1. **Simple Majority**: Entry committed when replicated to majority of servers (including leader).
> 2. **All Servers**: Wait for all non-failed servers to acknowledge.
> 3. **Tiered Commitment**: Different commitment levels for different durability guarantees.

> **Decision**: Use simple majority replication as the commit rule, with the **critical safety enhancement** from Section 5.4.2 of the Raft paper.

> **Rationale**: Simple majority ensures **availability** during partial failures while maintaining **safety**. The Raft paper's Section 5.4.2 adds a crucial rule: a leader can only commit entries from its current term once they're replicated to a majority. For entries from older terms, commitment happens indirectly when a later entry from the current term is committed. This avoids the "Figure 8" scenario where an old leader's entry could be committed then overwritten by a new leader with conflicting entries.

> **Consequences**:
>
> - **Positive**: Provides optimal availability (tolerates up to `⌊(n-1)/2⌋` failures); ensures safety with the Figure 8 prevention rule.
> - **Negative**: Requires tracking replication counts per entry; careful handling of term boundaries.
> - **Learning Benefit**: Demonstrates the subtlety of distributed consensus safety guarantees.

Commit Index Advancement Algorithm:

1. **Replication Tracking**: Leader maintains `matchIndex[i]` for each follower `i` (highest known replicated index).
2. **Majority Calculation**: After each successful `AppendEntries` response, update `matchIndex` and sort values.
3. **Commit Rule**: Set `commitIndex` to the median of `matchIndex` values (the majority replication point), **but only if**:
   - The entry at that index has term equal to leader's current term, OR
   - The entry is from an older term but a later entry from current term has been committed.
4. **Safety Invariant**: Never decrease `commitIndex`.

6.3 Common Pitfalls in Log Replication

⚠️ Pitfall: Off-by-One Errors in Log Indices

Description: Confusing 0-based vs. 1-based indexing, or mishandling the relationship between `prevLogIndex`, entry indices, and `commitIndex`. The Raft paper uses 1-based indexing for log entries (index 0 is a dummy entry with term 0).

Why It's Wrong: Causes followers to incorrectly accept/reject entries, leading to log divergence or infinite retry loops. For example, if a follower's log is empty and `prevLogIndex=0` is sent, the follower should check for a term 0 entry at index 0 (which exists as a sentinel), not try to access `log[0]` which might be out of bounds in a 0-based array.

How to Fix:

- Implement a `dummy entry` at index 0 with term 0 in all logs
- Consistently use 1-based indices in the protocol layer
- Create helper methods: `log.getTerm(index)` returns term 0 for index 0
- Validate: `prevLogIndex <= lastLogIndex` before accessing

⚠️ Pitfall: Applying Entries Before They Are Committed

Description: Applying entries to the state machine as soon as they're appended to the local log, rather than waiting for `commitIndex` to advance past them.

Why It's Wrong: Violates **State Machine Safety**. If an uncommitted entry is applied and the server crashes, after recovery it might revert to a state without that entry (if the entry wasn't committed before the crash), causing the state machine to diverge from other servers.

How to Fix:

- Maintain `lastApplied` index tracking last applied entry
- Periodically check if `commitIndex > lastApplied`
- Apply entries sequentially from `lastApplied+1` to `commitIndex`
- Only advance `lastApplied` after successful application

⚠️ Pitfall: Not Handling Log Inconsistencies During Recovery

Description: When a partitioned follower rejoins, its log may conflict with the leader's. Simply appending new entries without checking for conflicts or truncating divergent entries.

Why It's Wrong: Creates "holes" or duplicate entries in the log, violating the **Log Matching Property**. If follower has entries `[1:1, 2:1, 3:2]` and leader has `[1:1, 2:1, 3:3, 4:3]`, blindly appending leader's entries creates `[1:1, 2:1, 3:2, 3:3, 4:3]` with duplicate index 3.

How to Fix:

- Implement full **consistency check** using `prevLogIndex` / `prevLogTerm`
- When mismatch detected, follower must truncate log from `prevLogIndex+1`
- Leader must implement **log backtracking**: decrement `nextIndex` and retry
- Continue until a matching point is found

⚠️ Pitfall: Incorrect Commit Index Advancement from Previous Terms

Description: A leader committing entries from previous terms immediately upon replicating them to a majority, without the Section 5.4.2 safety rule.

Why It's Wrong: Leads to the **"Figure 8" scenario** from the Raft paper where an entry from an old term gets committed, then a new leader overwrites it with conflicting entries, violating the **Leader Completeness** property.

How to Fix:

- Implement the **commit rule**: A leader only counts replicas for commitment of an entry from its **current term**
- Once an entry from current term is committed, all preceding entries (including from older terms) become indirectly committed
- Track `matchIndex` per follower but only advance `commitIndex` based on current term entries

⚠️ Pitfall: Not Batching Entries for Efficiency

Description: Sending each log entry individually in separate `AppendEntries` RPCs rather than batching multiple entries.

Why It's Wrong: Severe performance degradation, especially under high load. Each RPC has network latency overhead, and followers process entries one at a time rather than in efficient batches.

How to Fix:

- Buffer incoming client commands briefly (e.g., up to 100ms or 100 entries)
- Send multiple entries in a single `AppendEntries` RPC
- Ensure `prevLogIndex` and `prevLogTerm` point to the entry before the first in the batch
- Process entire batch atomically in follower (all or nothing)

6.4 Implementation Guidance for Milestone 2

Technology Recommendations Table:

Component	Simple Option (Learning Focus)	Advanced Option (Production)
Log Storage	In-memory slice with mutex (`[]LogEntry`)	Persistent WAL with fsync, memory-mapped files
Entry Serialization	Go's `encoding/json`	Protocol Buffers, Cap'n Proto
Batch Processing	Fixed-size batches (e.g., 10 entries)	Dynamic batching based on size/time
Network Transport	HTTP/1.1 with JSON (already implemented)	gRPC stream for continuous replication

Recommended File/Module Structure:

```
project-root/
├── cmd/
│   ├── raft-node/          # Main executable
│   │   └── main.go
│   └── client/             # Test client
│       └── main.go
├── internal/
│   ├── raft/               # Core Raft implementation
│   │   ├── node.go         # RaftNode struct and main loop
│   │   ├── election.go     # Leader election logic (Milestone 1)
│   │   ├── replication.go  # Log replication logic (THIS MILESTONE)
│   │   ├── rpc_handlers.go # RequestVote and AppendEntries handlers
│   │   ├── state.go        # Persistent/volatile state management
│   │   └── safety.go       # Safety rules (Milestone 3)
│   ├── logstore/           # Log storage abstraction
│   │   ├── interface.go    # LogStore interface
│   │   ├── memory.go       # MemoryStore implementation
│   │   └── persistent.go   # File-based store (optional extension)
│   ├── statemachine/       # Application state machine
│   │   ├── interface.go    # StateMachine interface
│   │   └── kvstore.go      # Example key-value store
│   └── transport/          # Network communication
│       ├── interface.go    # Transport interface
│       ├── http.go         # HTTPTransport implementation
│       └── mock.go         # Mock transport for testing
└── pkg/
    └── types/              # Shared types
        ├── rpc.go          # RPC request/response structs
        └── log.go          # LogEntry and related types
```

Infrastructure Starter Code: Complete LogStore Interface and Implementation

```go
// File: internal/logstore/interface.go

package logstore

import "errors"

// LogStore defines the interface for storing Raft log entries.
// Implementations must guarantee thread safety.
type LogStore interface {
    // Append adds entries to the log starting at the given index.
    // Returns error if prevIndex doesn't match last index in store.
    Append(prevIndex int, prevTerm int, entries []LogEntry) error

    // Get returns the entry at the specified index (1-based).
    // Returns error if index is out of bounds.
    Get(index int) (LogEntry, error)

    // GetTerm returns the term of the entry at the specified index.
    // Returns 0 for index 0 (dummy entry).
    GetTerm(index int) int

    // LastIndex returns the index of the last entry in the log.
    // Returns 0 for empty log.
    LastIndex() int

    // LastTerm returns the term of the last entry in the log.
    // Returns 0 for empty log.
    LastTerm() int

    // Slice returns entries in the range [start, end] (inclusive).
    // If end is -1, returns all entries from start to end.
    Slice(start, end int) ([]LogEntry, error)

    // Truncate removes all entries from fromIndex onward (inclusive).
    // Returns error if fromIndex is greater than LastIndex()+1.
    Truncate(fromIndex int) error

    // Size returns the number of entries in the log (excluding dummy entry at index 0).
```

```
    Size() int
}

// Common errors
var (
    ErrIndexOutOfBounds = errors.New("log index out of bounds")
    ErrTermMismatch     = errors.New("previous log term mismatch")
    ErrIndexMismatch    = errors.New("previous log index mismatch")
    ErrInvalidIndex     = errors.New("invalid log index")
)
```

```go
// File: internal/logstore/memory.go

package logstore


import (

    "sync"

    "pkg/types"

)


// MemoryStore is an in-memory implementation of LogStore.

type MemoryStore struct {

    mu       sync.RWMutex

    entries  []types.LogEntry  // entries[0] is dummy entry at index 0

}


// NewMemoryStore creates an empty MemoryStore with dummy entry at index 0.

func NewMemoryStore() *MemoryStore {

    return &MemoryStore{

        entries: []types.LogEntry{{Index: 0, Term: 0}}, // Dummy entry

    }

}


// Append implements LogStore.Append.

func (s *MemoryStore) Append(prevIndex int, prevTerm int, entries []types.LogEntry) error {

    if len(entries) == 0 {

        return nil // Empty AppendEntries acts as heartbeat

    }



    s.mu.Lock()

    defer s.mu.Unlock()



    // Check if prevIndex matches our last index

    if prevIndex != s.LastIndex() {

        return ErrIndexMismatch

    }



    // Check if term at prevIndex matches

    if prevIndex > 0 && s.GetTerm(prevIndex) != prevTerm {

        return ErrTermMismatch
```

```
    }

    // Assign correct indices to new entries
    for i := range entries {
        entries[i].Index = prevIndex + i + 1
    }

    s.entries = append(s.entries, entries...)
    return nil
}

// Get implements LogStore.Get.
func (s *MemoryStore) Get(index int) (types.LogEntry, error) {
    s.mu.RLock()
    defer s.mu.RUnlock()

    if index < 0 || index >= len(s.entries) {
        return types.LogEntry{}, ErrIndexOutOfBounds
    }
    return s.entries[index], nil
}

// GetTerm implements LogStore.GetTerm.
func (s *MemoryStore) GetTerm(index int) int {
    if index == 0 {
        return 0 // Dummy entry term
    }

    s.mu.RLock()
    defer s.mu.RUnlock()

    if index < 0 || index >= len(s.entries) {
        return -1 // Invalid, but caller should handle
    }
    return s.entries[index].Term
}
```

```go
// LastIndex implements LogStore.LastIndex.
func (s *MemoryStore) LastIndex() int {
    s.mu.RLock()
    defer s.mu.RUnlock()
    return len(s.entries) - 1 // Minus dummy entry
}

// LastTerm implements LogStore.LastTerm.
func (s *MemoryStore) LastTerm() int {
    lastIdx := s.LastIndex()
    if lastIdx == 0 {
        return 0
    }
    return s.GetTerm(lastIdx)
}

// Slice implements LogStore.Slice.
func (s *MemoryStore) Slice(start, end int) ([]types.LogEntry, error) {
    s.mu.RLock()
    defer s.mu.RUnlock()

    if start < 1 || start >= len(s.entries) {
        return nil, ErrIndexOutOfBounds
    }

    if end == -1 {
        end = len(s.entries) - 1
    }

    if end < start || end >= len(s.entries) {
        return nil, ErrIndexOutOfBounds
    }

    result := make([]types.LogEntry, end-start+1)
    copy(result, s.entries[start:end+1])
    return result, nil
}
```

```
// Truncate implements LogStore.Truncate.
func (s *MemoryStore) Truncate(fromIndex int) error {
    s.mu.Lock()
    defer s.mu.Unlock()

    if fromIndex < 1 || fromIndex > len(s.entries) {
        return ErrInvalidIndex
    }

    s.entries = s.entries[:fromIndex]
    return nil
}

// Size implements LogStore.Size.
func (s *MemoryStore) Size() int {
    s.mu.RLock()
    defer s.mu.RUnlock()
    return len(s.entries) - 1 // Exclude dummy entry
}
```

Core Logic Skeleton Code: Leader Replication Loop

```go
// File: internal/raft/replication.go

package raft

import (
    "log"
    "sync"
    "time"
    "pkg/types"
)

// startReplicationLoop begins the leader's replication goroutine.
// Runs until the node is no longer leader.
func (rn *RaftNode) startReplicationLoop() {
    rn.mu.Lock()
    state := rn.state
    rn.mu.Unlock()

    if state != Leader {
        return // Should not be called when not leader
    }

    // Initialize leader volatile state
    rn.mu.Lock()
    lastIdx := rn.logStore.LastIndex()
    for i := range rn.peers {
        rn.nextIndex[i] = lastIdx + 1
        rn.matchIndex[i] = 0
    }
    rn.mu.Unlock()

    // Start heartbeat ticker
    heartbeatTicker := time.NewTicker(50 * time.Millisecond)
    defer heartbeatTicker.Stop()

    for {
        select {
        case <-heartbeatTicker.C:
```

```
            rn.broadcastHeartbeats()


        case <-rn.replicationTrigger:

            rn.broadcastAppendEntries()


        case <-rn.stopCh:

            return

        }

    }

}


// broadcastHeartbeats sends empty AppendEntries RPCs to all followers.

func (rn *RaftNode) broadcastHeartbeats() {

    rn.mu.Lock()


    // TODO 1: Check if node is still leader; return if not


    // TODO 2: For each peer (excluding self):

    //   - Prepare AppendEntriesArgs with:

    //     * Term: rn.currentTerm

    //     * LeaderID: rn.id

    //     * PrevLogIndex: rn.nextIndex[peer] - 1

    //     * PrevLogTerm: term at PrevLogIndex (use logStore.GetTerm)

    //     * Entries: empty slice (heartbeat)

    //     * LeaderCommit: rn.commitIndex

    //   - Send RPC asynchronously (goroutine)


    rn.mu.Unlock()

}


// broadcastAppendEntries sends new entries to followers that need them.

func (rn *RaftNode) broadcastAppendEntries() {

    rn.mu.Lock()


    // TODO 3: Check if node is still leader; return if not


    // TODO 4: For each peer (excluding self):
```

```
    //   - If rn.nextIndex[peer] <= rn.logStore.LastIndex():
    //     * Prepare entries to send: slice from rn.nextIndex[peer] to end
    //     * Prepare AppendEntriesArgs similar to broadcastHeartbeats but with entries
    //     * Send RPC asynchronously
    //   - Else: send heartbeat (no new entries for this peer)

    rn.mu.Unlock()
}

// handleAppendEntriesResult processes a response from a follower.
func (rn *RaftNode) handleAppendEntriesResult(peerID int, args types.AppendEntriesArgs, reply types.AppendEntriesReply) {
    rn.mu.Lock()
    defer rn.mu.Unlock()

    // TODO 5: If reply.Term > rn.currentTerm:
    //   - Convert to follower
    //   - Update rn.currentTerm = reply.Term
    //   - Reset election timer
    //   - Return

    // TODO 6: If reply.Success == true:
    //   - Update rn.nextIndex[peerID] = args.PrevLogIndex + len(args.Entries) + 1
    //   - Update rn.matchIndex[peerID] = rn.nextIndex[peerID] - 1
    //   - Update commit index (call rn.advanceCommitIndex())

    // TODO 7: If reply.Success == false (log inconsistency):
    //   - Decrement rn.nextIndex[peerID] by 1
    //   - Ensure it doesn't go below 1
    //   - Trigger another AppendEntries to this peer
}
```

Core Logic Skeleton Code: AppendEntries RPC Handler

GO

```go
// File: internal/raft/rpc_handlers.go

package raft

import "pkg/types"

// handleAppendEntries processes an incoming AppendEntries RPC (both heartbeat and replication).
func (rn *RaftNode) handleAppendEntries(args types.AppendEntriesArgs) types.AppendEntriesReply {
    rn.mu.Lock()
    defer rn.mu.Unlock()

    reply := types.AppendEntriesReply{}

    // TODO 1: If args.Term < rn.currentTerm:
    //   - Set reply.Term = rn.currentTerm
    //   - Set reply.Success = false
    //   - Return reply

    // TODO 2: Reset election timer (leader heartbeat received)
    //   - Call rn.resetElectionTimeout()

    // TODO 3: If args.Term > rn.currentTerm:
    //   - Update rn.currentTerm = args.Term
    //   - Convert to follower (rn.state = Follower)
    //   - Clear rn.votedFor

    // TODO 4: Consistency check:
    //   - If args.PrevLogIndex > rn.logStore.LastIndex():
    //     * Set reply.Success = false
    //     * Set reply.Term = rn.currentTerm
    //     * Return reply
    //   - If args.PrevLogIndex >= 1 and rn.logStore.GetTerm(args.PrevLogIndex) != args.PrevLogTerm:
    //     * Set reply.Success = false
    //     * Set reply.Term = rn.currentTerm
    //     * Return reply

    // TODO 5: If we reach here, logs match at PrevLogIndex
```

```
    // TODO 6: Handle entry appending:
    //   - If len(args.Entries) > 0:
    //     * Find conflict point: iterate through args.Entries, for each entry at index i:
    //       If (args.PrevLogIndex + 1 + i) <= rn.logStore.LastIndex() and
    //          rn.logStore.GetTerm(args.PrevLogIndex + 1 + i) != entry.Term:
    //         - Truncate log from conflict point (rn.logStore.Truncate)
    //         - Break
    //     * Append new entries (rn.logStore.Append)

    // TODO 7: Update commit index:
    //   - If args.LeaderCommit > rn.commitIndex:
    //     * Set rn.commitIndex = min(args.LeaderCommit, rn.logStore.LastIndex())
    //     * Trigger application of committed entries (call rn.applyCommittedEntries())

    // TODO 8: Set reply.Success = true
    // TODO 9: Set reply.Term = rn.currentTerm

    return reply
}

// advanceCommitIndex updates the commit index based on matchIndex values.
// Implements the safety rule from Section 5.4.2 of the Raft paper.
func (rn *RaftNode) advanceCommitIndex() {
    // TODO 10: Only leaders should advance commit index
    //   - Check if rn.state != Leader, return early if true

    // TODO 11: Create a copy of matchIndex values
    //   - Include leader's own log (rn.logStore.LastIndex())
    //   - Sort the values

    // TODO 12: Find the median (N/2 + 1) where N = len(rn.peers)
    //   - This is the majority replication point

    // TODO 13: Safety check: Only advance if entry at median index has term == rn.currentTerm
    //   - If rn.logStore.GetTerm(medianIndex) == rn.currentTerm:
    //     * Set rn.commitIndex = max(rn.commitIndex, medianIndex)
```

```
    // TODO 14: Trigger application of newly committed entries
    //   - Call rn.applyCommittedEntries()
}

// applyCommittedEntries applies entries between lastApplied and commitIndex to state machine.
func (rn *RaftNode) applyCommittedEntries() {
    // TODO 15: While rn.lastApplied < rn.commitIndex:
    //   - Get next entry: rn.logStore.Get(rn.lastApplied + 1)
    //   - Apply to state machine: rn.stateMachine.Apply(entry.Command)
    //   - Increment rn.lastApplied
    //   - Notify any waiting clients (if this was a client command)
}
```

Language-Specific Hints for Go:

1. **Concurrency Pattern**: Use a separate goroutine for the replication loop, triggered by a `replicationTrigger` channel. This follows Go's "share memory by communicating" principle.
2. **Batch Processing**: Use Go slices efficiently: `entries := rn.logStore.Slice(nextIndex, -1)` returns all entries from `nextIndex` to end.
3. **RPC Timeouts**: Always use context with timeout for RPC calls: `ctx, cancel := context.WithTimeout(context.Background(), 100*time.Millisecond)`.
4. **Memory Efficiency**: When sending large batches, consider compressing entries or using streaming RPCs for production systems.
5. **Testing**: Use Go's `testing` package with table-driven tests for different log inconsistency scenarios.

Milestone 2 Checkpoint:

To verify your implementation works correctly:

1. **Run the Basic Test**: Start a 3-node cluster and send a single command:

```bash
go run cmd/raft-node/main.go --id=1 --peers=1,2,3 &
go run cmd/raft-node/main.go --id=2 --peers=1,2,3 &
go run cmd/raft-node/main.go --id=3 --peers=1,2,3 &
echo "SET key1 value1" | nc localhost 8081  # Send to leader
```

2. **Expected Behavior**:
   - One node becomes leader (logs "became leader at term X")
   - Leader receives command and appends to log
   - Leader sends AppendEntries to followers
   - Followers accept and append entries
   - Leader commits entry after majority acknowledgment
   - All nodes apply entry to state machine
   - Leader responds to client with success
3. **Verification Commands**:

```bash
# Check logs on each node
curl http://localhost:8081/debug/log | jq '.entries'
curl http://localhost:8082/debug/log | jq '.entries'
curl http://localhost:8083/debug/log | jq '.entries'

# Should show identical logs across all nodes
```

4. **Signs of Problems**:
   - **No replication**: Check `PrevLogIndex` / `PrevLogTerm` logic
   - **Infinite retries**: Likely off-by-one error in `nextIndex` management
   - **Entries not committed**: Check majority calculation and safety rule
   - **State machines diverge**: Verify `applyCommittedEntries` only applies committed entries

Debugging Tips Table:

Symptom	Likely Cause	How to Diagnose	Fix
"Follower rejects all AppendEntries"	`PrevLogIndex` / `PrevLogTerm` mismatch	Log follower's last index/term vs. leader's `prevLogIndex` / `prevLogTerm`	Ensure leader's `nextIndex` starts at `lastIndex+1` after election
"Leader never advances commitIndex"	Majority calculation wrong	Check `matchIndex` values; verify quorum size calculation	Use `(len(peers)/2)+1` for majority; include leader in count
"Entries applied out of order"	Applying uncommitted entries	Check `lastApplied` vs `commitIndex` before applying	Only apply when `commitIndex > lastApplied`
"Log grows without bound on follower mismatch"	Not truncating on conflict	Log before/after AppendEntries handling	Implement `Truncate` in `handleAppendEntries`
"Client command hangs forever"	Leader crashed before committing	Check leader logs; verify commit index advancement	Implement client timeout and retry with idempotent commands

7. Component Design: Safety Properties (Milestone 3)

> **Milestone(s):** 3 (Safety Properties)

While leader election and log replication provide the *mechanisms* for consensus, the **Safety Properties** are the *guarantees* that make Raft trustworthy. This milestone ensures that despite failures, network partitions, and concurrent operations, the system never violates its core correctness promises. It's what transforms a collection of cooperating nodes into a reliable foundation for building distributed systems.

7.1 Mental Model: The Chain of Custody for Legal Evidence

Imagine a criminal investigation where physical evidence (a weapon, a document) must move between police stations, forensic labs, and courtrooms. To be admissible, this evidence requires an **unbroken chain of custody**: every transfer is logged, every handler signs for it, and any gap or alteration invalidates the entire case.

This is precisely Raft's safety model:

- **The Log Entries are the Evidence:** Each command that modifies the state machine is a piece of evidence that must be preserved.
- **Index and Term are the Logbook:** The `Index` (sequence number) and `Term` (epoch) form the tamper-proof log of who had the evidence and when.
- **Leaders are the Authorized Couriers:** Only the current leader (elected for a specific term) can introduce new evidence into the system.

- **Quorum Sign-Off is the Verification:** A majority of nodes (a quorum) must acknowledge receiving the evidence before it's considered permanently recorded.
- **The Commit Index is the Court's Seal:** Once evidence is "committed," it's irrevocably part of the official record and will be presented (applied) to the state machine.

The safety properties ensure that:

1. **No evidence is lost** once committed (Leader Completeness).
2. **No two conflicting chains of evidence** are accepted as valid (Election Safety).
3. **Every court (node) applies evidence in the exact same order** (State Machine Safety).

Just as a break in the chain of custody compromises a legal case, violating any Raft safety property compromises the entire distributed system's correctness.

7.2 Architecture Decision Records for Safety

The Raft paper's Section 5.4.2 describes a subtle but critical safety issue illustrated in **Figure 8**, where a leader could commit an entry from a previous term that might later be overwritten by a newer leader. The protocol must include a specific rule to prevent this.

> **Decision: Leader-Only Commitment Rule for Safety**
>
> - **Context**: After implementing leader election and log replication, we must ensure the **State Machine Safety Property**: if a server has applied a log entry at a given index to its state machine, no other server will ever apply a different log entry for the same index. The Figure 8 scenario shows this can be violated if a leader commits an entry from an old term without special handling.
> - **Options Considered**:
>   1. **Commit Any Entry on Majority Replication**: A leader considers an entry committed once it's stored on a majority of servers, regardless of the entry's term. This is simple but leads to the Figure 8 problem where an old entry could be committed and then overwritten.
>   2. **Leader-Only Term Commitment (Raft Paper Rule)**: A leader only counts replicas for entries from its *own term* when determining commit progress. Once an entry from its term is committed, all prior entries are indirectly committed via the Log Matching Property.
>   3. **Term-Index Pair Majority**: Require that an entry is committed only when a majority of servers have not only the entry but also all preceding entries. This is complex to track and compute.
> - **Decision**: Implement **Option 2: Leader-Only Term Commitment** as specified in the Raft paper (Section 5.4.2). This is the standard Raft safety rule.
> - **Rationale**:
>   - **Correctness**: It provably prevents the Figure 8 scenario by ensuring a leader's commit index only advances based on entries from its current term. This guarantees that any committed entry will survive future leader changes.
>   - **Simplicity**: The rule is simple to implement—when updating the commit index, only consider entries where `log[i].Term == currentTerm`. The Log Matching Property then ensures all preceding entries are also committed.
>   - **Standardization**: This is the universally accepted solution in all Raft implementations, ensuring compatibility with the reference algorithm and educational materials.
> - **Consequences**:
>   - **Slightly Slower Commitment in Some Cases**: An entry from a previous term replicated to a majority might not become committed until a later leader from a new term replicates one of its own entries. This is acceptable because safety trumps liveness in consensus algorithms.
>   - **Implementation Complexity**: Requires tracking which entries belong to the leader's term and carefully updating `commitIndex` only when those entries achieve majority replication.

Option	Pros	Cons	Chosen?
Commit Any Entry on Majority	Simple logic; fast commitment of old entries	Violates safety in Figure 8 scenario; allows committed entries to be lost	❌
Leader-Only Term Commitment (Raft Rule)	**Provably safe; simple to implement; standard approach**	Slightly delays commitment of old-term entries until leader adds new entry	✅
Term-Index Pair Majority	Theoretically sound; allows earlier commitment	Complex implementation; non-standard; difficult to reason about	❌

The implementation of this rule occurs in the `advanceCommitIndex()` method, which leaders call after receiving successful `AppendEntries` responses. The rule is: for each index `N` greater than the current `commitIndex`, if a majority of `matchIndex[peer] >= N` **and** `log[N].Term == currentTerm`, then `commitIndex` can be advanced to `N`.

7.3 Common Pitfalls in Safety Properties

Implementing safety properties requires meticulous attention to detail. Here are the most common mistakes:

⚠️ Pitfall 1: The "Figure 8" Scenario - Committing Old Term Entries

- **Description**: A leader replicates an entry from term 2 to a majority but crashes before committing it. A new leader (term 3) is elected, overwrites that entry, and might commit a different entry at the same index, violating safety.
- **Why It's Wrong**: If the old term 2 entry was considered committed by the first leader (using simple majority rule), then having a different entry at the same index applied to state machines breaks the State Machine Safety Property.
- **How to Fix**: Implement the **Leader-Only Term Commitment** rule exactly as described in ADR 7.2. In `advanceCommitIndex()`, only consider entries where `log[i].Term == currentTerm` when checking for majority replication.

⚠️ Pitfall 2: Violating Leader Completeness During Election

- **Description**: During `handleRequestVote()`, a node grants a vote to a candidate whose log is *less complete* (has fewer committed entries) than its own, potentially electing a leader that will overwrite committed entries.
- **Why It's Wrong**: This breaks the **Leader Completeness Property** (Raft Safety Property #2): if a log entry is committed in a given term, then that entry will be present in the logs of the leaders for all higher-numbered terms.
- **How to Fix**: Strictly implement the **log comparison** rule in `RequestVote` RPC handling. A candidate's log is "more up-to-date" if:
  1. The last term in the candidate's log is **greater than** the receiver's last term, OR
  2. The terms are equal, but the candidate's log is **at least as long** as the receiver's log. Only grant a vote if the candidate's log is at least as up-to-date. This ensures elected leaders contain all committed entries.

⚠️ Pitfall 3: Applying Entries Before They Are Committed

- **Description**: The `applyCommittedEntries()` loop applies entries to the state machine as soon as `lastApplied < commitIndex`, but if `commitIndex` is incorrectly advanced (e.g., due to a bug in `advanceCommitIndex()`), uncommitted entries might be applied prematurely.
- **Why It's Wrong**: This violates the core guarantee of consensus: only committed (permanently agreed-upon) entries should affect the state machine. Premature application could lead to inconsistent state across nodes if those entries are later overwritten.
- **How to Fix**: Ensure `commitIndex` advancement follows the safety rule strictly. Additionally, consider adding an **invariant check**: before applying an entry, verify that it's actually in the log and has the expected index and term. While Raft doesn't require this check, it's good defensive programming for debugging.

⚠️ Pitfall 4: Incorrectly Handling Empty Logs During Election

- **Description**: When a node with an empty log (no entries) receives a `RequestVote` RPC, it incorrectly rejects candidates because of off-by-one errors in log comparison logic.
- **Why It's Wrong**: A node with an empty log should vote for any candidate (assuming other conditions like term are met), because any log is at least as up-to-date as an empty log. Failing to do so can prevent elections in initial cluster startup or after catastrophic failures.
- **How to Fix**: Handle the empty log as a special case in log comparison. An empty log has `lastLogIndex = 0` and `lastLogTerm = 0`. The comparison rules still work mathematically (0 ≤ any term, 0 ≤ any index), but ensure your implementation doesn't have index-off-by-one errors where you try to access `log[0]` when the log is empty.

7.4 Implementation Guidance for Milestone 3

This milestone focuses on enhancing the existing leader election and log replication components with the safety rules. The primary additions are the corrected `advanceCommitIndex()` logic and ensuring `RequestVote` implements proper log comparison.

A. Technology Recommendations Table

Component	Simple Option	Advanced Option
Safety Rule Enforcement	Direct implementation in `advanceCommitIndex()` with majority calculation	Configurable safety rules for experimentation
Log Comparison	Direct term/index comparison in `handleRequestVote`	Abstract `LogComparer` interface for testing edge cases
State Machine Application	Direct function call in main loop	Separate application goroutine with channel for committed entries

B. Recommended File/Module Structure

The safety enhancements should be integrated into the existing Raft node implementation:

```
project-root/
  internal/raft/
    node.go              # Updated with safety rules in advanceCommitIndex
    election.go          # Updated with strict log comparison in handleRequestVote
    replication.go       # Contains advanceCommitIndex and applyCommittedEntries
    state.go             # Core data structures (unchanged)
    transport.go         # Unchanged
  internal/statemachine/
    interface.go         # State machine interface
    kv.go                # Example key-value state machine
```

C. Core Logic Skeleton Code

Below are the critical methods that must be updated or implemented with safety in mind. The TODO comments map directly to the numbered steps in the algorithm descriptions.

```go
// In replication.go

// advanceCommitIndex updates the leader's commit index based on matchIndex values.
// This implements the critical safety rule: only entries from the leader's current term
// are considered when advancing the commit index.
func (rn *RaftNode) advanceCommitIndex() {
    rn.mu.Lock()
    defer rn.mu.Unlock()

    // Safety check: only leaders should advance commit index
    if rn.state != Leader {
        return
    }

    // TODO 1: Create a slice of all matchIndex values (including leader's own index)
    // Hint: The leader's own matchIndex is always the last log index.

    // TODO 2: Sort the matchIndex slice to find the median (majority) value
    // Explanation: If we have N nodes, we need a majority (N/2 + 1).
    // After sorting, the value at position (N/2) is the majority commit point candidate.

    // TODO 3: Iterate from current commitIndex + 1 up to the majority candidate index
    // For each index N in that range:
    //   - Check if log[N].Term == rn.currentTerm (SAFETY RULE: only commit entries from current term)
    //   - If true, set rn.commitIndex = N
    //   - Otherwise, stop (cannot commit entries from older terms beyond this point)

    // TODO 4: After updating commitIndex, trigger application of newly committed entries
    // Hint: You can call rn.applyCommittedEntries() or signal a condition variable.
}

// applyCommittedEntries applies all committed entries that haven't been applied yet
// to the state machine. This must be called after commitIndex advances.
func (rn *RaftNode) applyCommittedEntries() {
    rn.mu.Lock()
    defer rn.mu.Unlock()

    // TODO 1: Determine the range of entries to apply
    // Start index: rn.lastApplied + 1
    // End index: rn.commitIndex (inclusive)
```

```
    // TODO 2: For each entry in that range (in order):
    //   - Apply the entry.Command to the state machine (rn.stateMachine.Apply(...))
    //   - Update rn.lastApplied to the index of the applied entry

    // TODO 3: Ensure all applications are done while holding the lock or
    // use careful synchronization if applying in a separate goroutine

    // TODO 4: Log the application for debugging (optional but recommended)
}
```

```go
// In election.go

// handleRequestVote processes an incoming RequestVote RPC.
// The log comparison logic here is critical for Leader Completeness property.
func (rn *RaftNode) handleRequestVote(args RequestVoteArgs) RequestVoteReply {
    rn.mu.Lock()
    defer rn.mu.Unlock()

    reply := RequestVoteReply{Term: rn.currentTerm, VoteGranted: false}

    // TODO 1: If args.Term < rn.currentTerm, return immediately with VoteGranted=false
    // (already implemented in previous milestone)

    // TODO 2: If args.Term > rn.currentTerm, step down to follower and update currentTerm
    // (already implemented in previous milestone)

    // TODO 3: Check if we've already voted for someone else this term
    // Condition: (rn.votedFor == -1 || rn.votedFor == args.CandidateID)

    // TODO 4: CRITICAL SAFETY CHECK: Is candidate's log at least as up-to-date as ours?
    // Implement the log comparison rule:
    //   a. Get our last log term and index
    //   b. Compare last log terms:
    //        - If args.LastLogTerm > ourLastLogTerm → candidate's log is more up-to-date
    //        - If args.LastLogTerm < ourLastLogTerm → candidate's log is less up-to-date
    //        - If terms are equal, compare lengths:
    //            - If args.LastLogIndex >= ourLastLogIndex → candidate's log is at least as up-to-date
    //            - Otherwise → candidate's log is less up-to-date
    //   c. Only grant vote if candidate's log is at least as up-to-date

    // TODO 5: If all checks pass, grant vote:
    //   - Set rn.votedFor = args.CandidateID
    //   - Reset election timeout (since we acknowledged a legitimate candidate)
    //   - Set reply.VoteGranted = true

    return reply
}
```

D. Language-Specific Hints for Go

1. **Sorting Helper**: For `advanceCommitIndex`, use `sort.Ints(matchIndexCopy)` to sort the matchIndex values.
2. **Majority Calculation**: For N nodes, majority index is `N/2` in 0-indexed sorted array (Go integer division truncates).

3. **Log Access Safety**: Always check log bounds before accessing `rn.log[index]`. Consider helper methods like `logTermAt(index int)` that returns 0 for indices before the start of the log.
4. **State Machine Interface**: Define a simple interface for the state machine:

```go
type StateMachine interface {
    Apply(command interface{}) (interface{}, error)
}
```

5. **Concurrent Application**: If applying entries in a separate goroutine for performance, use a channel to send committed entries and update `lastApplied` with proper locking.

E. Milestone Checkpoint

To verify your safety implementation is correct:

1. **Run a basic test** that creates a 3-node cluster and submits several commands:

```
go test -v -run TestSafetyBasic ./internal/raft/
```

Expected: All nodes should have identical logs after commands are committed.

2. **Simulate the Figure 8 scenario** manually or with a test:
   - Start a 5-node cluster.
   - Have leader S1 (term 2) replicate an entry but crash before committing it.
   - Elect S5 (term 3) as new leader.
   - Verify that S5 does *not* commit a conflicting entry at the same index until it replicates an entry from its own term (term 3).
   - Expected: The system should not apply conflicting entries at the same index.
3. **Check the invariant** programmatically: After each test, verify that for all nodes:
   - `commitIndex` ≤ last log index
   - `lastApplied` ≤ `commitIndex`
   - If two nodes have `commitIndex = N`, their logs are identical up to index N.
4. **Signs of Problems**:
   - **Symptom**: Different nodes apply different commands at the same index. **Diagnosis**: Check `advanceCommitIndex` logic—are you only committing entries from current term?
   - **Symptom**: A leader gets elected but missing previously committed entries. **Diagnosis**: Check `handleRequestVote` log comparison—are you correctly comparing last log term and index?
   - **Symptom**: `commitIndex` advances beyond what's replicated to majority. **Diagnosis**: Verify majority calculation and matchIndex tracking.

F. Debugging Tips

Symptom	Likely Cause	How to Diagnose	Fix
Different state machine outputs on different nodes	State Machine Safety violation (different entries applied at same index)	Add logging to `applyCommittedEntries` to see which entries each node applies. Check if `commitIndex` advancement follows the safety rule.	Ensure `advanceCommitIndex` only commits entries from leader's current term.
Newly elected leader overwrites committed entries	Leader Completeness violation (elected leader missing committed entries)	Log the `RequestVote` RPCs and check log comparison logic. Verify the elected leader's log before and after election.	Fix `handleRequestVote` to require candidate's log to be at least as up-to-date.
commitIndex never advances even with majority replication	Bug in `advanceCommitIndex` logic or matchIndex tracking	Log matchIndex values and currentTerm. Check if you're filtering by term correctly.	Ensure you're including the leader's own log in majority calculation and checking `log[N].Term == currentTerm`.
Split brain (two leaders in same term)	Election Safety violation	Check if `votedFor` is being persisted correctly. Check if nodes are updating `currentTerm` properly when receiving RPCs with higher term.	Ensure all RPC handlers that see a higher term step down to follower and update `currentTerm`.

Add structured logging to critical safety points:

- Log when `commitIndex` is updated (with the index and term of the entry being committed).
- Log when granting a vote (with candidate's and own last log term/index).
- Log when applying an entry to the state machine.

These logs will be invaluable when debugging safety violations, allowing you to trace exactly when and why incorrect decisions were made.

8. Component Design: Cluster Membership Changes (Milestone 4)

> **Milestone(s):** 4 (Cluster Membership Changes)

Up to this point, our Raft cluster has operated with a static configuration—a fixed set of servers known at startup. However, real-world systems must evolve: hardware fails and needs replacement, capacity requirements change, or clusters need scaling. **Cluster Membership Changes** address how to safely reconfigure a running Raft cluster without violating the safety properties we've worked so hard to establish. This is notoriously challenging because any change to the voting membership can create split-brain scenarios where two disjoint majorities each believe they have the authority to make decisions. This section introduces the **Joint Consensus** protocol, the standard solution described in the Raft paper, which provides a safe transition between configurations while maintaining availability.

8.1 Mental Model: Changing the Locks on a Bank Vault

Imagine a bank vault secured by a complex lock requiring multiple keys. The bank's policy states that the vault can only be opened when a majority of the current keyholders are present and agree. Now, the bank needs to change the keyholders—perhaps adding new trustees and retiring others. The naïve approach would be to simply issue new keys and revoke old ones, but during the transition, what if a group with only old keys and a group with only new keys each try to open the vault simultaneously? Both groups might believe they have a majority (if we don't coordinate properly), leading to the vault being opened without proper consensus—a clear safety violation.

The **Joint Consensus** protocol solves this by creating a temporary "dual-key" system. During the transition:

1. **Old and New Configuration Coexist**: For a period, both the old set of keys *and* the new set are valid.
2. **Dual Majority Requirement**: To open the vault, you now need a majority from the *old* keyholders **AND** a majority from the *new* keyholders. This ensures that during the transition, no group using only old keys or only new keys can act alone.
3. **Gradual Phase-Out**: Only after this intermediate configuration is fully established do we transition to requiring only a majority of the new keyholders, at which point the old keys are invalidated.

In Raft terms, the "keys" are the voting members of the cluster, and "opening the vault" corresponds to committing log entries (including the configuration change entry itself). The joint consensus protocol ensures that during a membership change, there is never a moment where two disjoint majorities could each elect a leader and commit conflicting entries, which would break the **State Machine Safety** property.

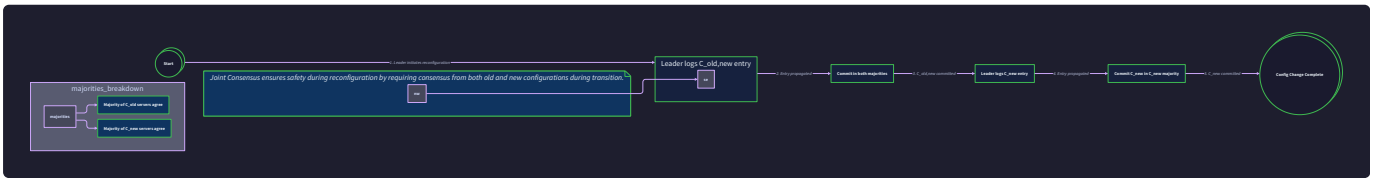
8.2 Architecture Decision Records for Membership

Decision: Use Joint Consensus for Configuration Changes

- **Context**: The cluster must be able to add or remove servers dynamically while maintaining the Raft safety guarantees. A simple approach of directly switching from one configuration to another risks creating disjoint majorities—a scenario where the old majority and new majority are different sets of servers that don't overlap, potentially allowing two leaders to be elected in the same term.
- **Options Considered**:
  1. **Direct Single-Server Changes**: Allow only one server to be added or removed at a time, with the new configuration taking effect immediately on the leader and being replicated via a special log entry. This is simpler but still vulnerable to the disjoint majority problem during the replication window.
  2. **Joint Consensus (Two-Phase)**: Introduce an intermediate configuration (`Cold,new`) that includes both old and new servers. Decisions (leader election, entry commitment) require majorities from both the old and new sets. After `Cold,new` is committed, the leader then replicates the final new configuration (`Cnew`). This guarantees safety but adds complexity.
  3. **External Coordination Service**: Use an external service (like ZooKeeper) to coordinate configuration changes, ensuring only one configuration is active at a time. This offloads complexity but introduces a new dependency and potential single point of failure.
- **Decision**: Implement **Joint Consensus** as specified in the Raft paper (Section 6).
- **Rationale**: Joint Consensus is the only option that provides formal safety guarantees during arbitrary membership changes without external dependencies. While more complex than single-server changes, it correctly handles all edge cases, including simultaneous configuration changes and leader failures during transition. The Raft paper's detailed specification provides a clear roadmap for implementation, making it suitable for an educational project where understanding safety is paramount.
- **Consequences**: The implementation must handle tracking two configurations simultaneously, calculating majorities across both sets, and ensuring servers in both configurations participate in the protocol. This increases the complexity of the election and commitment logic. However, it yields a production-grade membership change mechanism that can safely add or remove multiple servers at once.

Option	Pros	Cons	Chosen?
Direct Single-Server Changes	Simpler to implement; fewer states to track	Vulnerable to disjoint majorities during replication; only safe for single-server changes	No
Joint Consensus	**Formally safe for arbitrary changes; no external dependencies**	**More complex; requires dual-configuration tracking**	**Yes**
External Coordination Service	Simplifies Raft core logic; can use battle-tested coordination	Introduces new failure modes and dependency; contradicts goal of self-contained consensus	No

To implement joint consensus, we extend our data model with configuration tracking and update our RPC handlers and leader logic to respect the dual-majority rules during transitions. The process follows a precise sequence, visualized in the joint consensus flowchart:



8.3 Common Pitfalls in Membership Changes

⚠️ Pitfall: Disjoint Majorities Leading to Dual Leaders

- **Description**: During a configuration change from `Cold` (servers {1,2,3}) to `Cnew` (servers {3,4,5}), if the leader replicates `Cnew` directly without joint consensus, a network partition could isolate server 3 (the only overlapping member). The old majority {1,2} could elect a leader using `Cold` , while the new majority {4,5} could elect a different leader using `Cnew` . Both leaders could then commit conflicting entries.
- **Why It's Wrong**: This violates **Election Safety** (two leaders in same term) and **State Machine Safety** (diverging logs). The root cause is that the two configurations have disjoint majorities.
- **How to Fix**: Always use joint consensus. The intermediate configuration `Cold,new` includes all servers from both sets ({1,2,3,4,5}). Because any majority in `Cold,new` must include at least one server from `Cold` and one from `Cnew` (since majorities must overlap), disjoint majorities become impossible.

⚠️ Pitfall: Stuck in Intermediate Configuration

- **Description**: After a leader logs the joint consensus entry `C_old,new`, it crashes before committing it. A new leader from either the old or new configuration might not have the `C_old,new` entry and could attempt to replicate a different configuration change, leading to a deadlock where no configuration can achieve a majority.
- **Why It's Wrong**: The cluster cannot make progress because configurations are inconsistent across servers, and no leader can get elected with a complete log that includes the necessary configuration entry.
- **How to Fix**: Implement the "no new configuration until previous one committed" rule from the Raft paper. A leader may not replicate a new configuration entry (`C_old,new` or `C_new`) until the previous one is committed. Additionally, when a server restarts or rejoins, it must use its latest configuration (persisted with the log) for all RPCs, ensuring consistency.

⚠️ Pitfall: Not Isolating Removed Nodes

- **Description**: After a configuration change removes a server, that server continues to start elections and send RPCs, potentially disrupting the cluster because it is no longer part of the majority calculations.
- **Why It's Wrong**: A removed server that doesn't know it's removed could become a candidate, increment its term, and communicate with other servers, causing them to update to higher terms and triggering unnecessary leader elections.
- **How to Fix**: When a server learns of a new configuration `C_new` that doesn't include itself (via a committed configuration entry), it must transition to a **non-voting** state. It should stop its election timer, reject any incoming RequestVote RPCs, and should not start elections. It may continue to receive AppendEntries from the leader to stay up-to-date (acting as a learner) but does not participate in consensus.

⚠️ Pitfall: Adding a New Node That Is Not Caught Up

- **Description**: A new server is added to the configuration and immediately counted in majority calculations. If it has an empty log, it could cause the leader's commit index to stall because the new server's log is far behind, reducing availability.
- **Why It's Wrong**: The Raft paper's safety proof assumes that any server in the configuration has a log that is sufficiently up-to-date to participate in elections and commitment. A severely lagging server could prevent commitment of new entries because it cannot replicate quickly enough.
- **How to Fix**: Implement the **new server catch-up** mechanism. A new server is added as a **non-voting member** (learner) initially. The leader replicates entries to it until its log is sufficiently caught up (e.g., within a certain number of entries of the leader's log). Only then does the leader replicate a configuration change that includes the new server as a full voting member. This ensures availability is not impacted during the catch-up phase.

8.4 Implementation Guidance for Milestone 4

A. Technology Recommendations Table

Component	Simple Option	Advanced Option
Configuration Storage	Store configuration entries directly in the log (`LogEntry.Command` holds config)	Dedicated configuration manager with versioning
Majority Calculator	Function that iterates over configuration sets	Optimized bitmask representation for fast majority checks
Learner Mode	Basic flag to disable voting	Full learner state with progress tracking

B. Recommended File/Module Structure

```
project-root/
  cmd/server/main.go                  # Entry point with config parsing
  internal/raft/
    node.go                           # Main Raft node logic (extends existing)
    state.go                          # State definitions (NodeState, etc.)
    log_store.go                      # LogStore interface and MemoryStore
    config.go                         # NEW: Configuration structures and logic
    consensus.go                      # Election and replication logic (updated)
    persistence.go                    # Persistent state handling
  internal/transport/
    http.go                           # HTTP transport (HTTPServer)
  internal/statemachine/
    interface.go                      # StateMachine interface
    kv.go                             # Example key-value state machine
  test/
    membership_test.go                # Tests for membership changes
```

C. Infrastructure Starter Code

We need to define configuration structures and extend the log entry to support configuration commands.

GO

```go
// internal/raft/config.go

package raft

import (
    "encoding/json"
    "fmt"
    "sort"
)

// Configuration represents a cluster configuration.
// It is stored as the Command in a LogEntry when Type is ConfigEntry.
type Configuration struct {
    // Servers is a list of node IDs that are voting members in this configuration.
    Servers []int `json:"servers"`
    // Index is the log index at which this configuration takes effect.
    // This field is set when the configuration is created by the leader.
    Index int `json:"index,omitempty"`
    // Term is the term when this configuration was created.
    Term int `json:"term,omitempty"`
}

// ConfigState represents the node's view of current and upcoming configurations.
type ConfigState struct {
    // Current configuration (committed and in effect).
    Current Configuration
    // Pending configuration (joint consensus intermediate or new configuration not yet committed).
    // If nil, no configuration change is in progress.
    Pending *Configuration
}

// IsJoint returns true if the node is in a joint consensus state.
func (cs *ConfigState) IsJoint() bool {
    return cs.Pending != nil
}

// ContainsServer checks if a server ID is a voting member in the current effective configuration.
// During joint consensus, a server is a voting member if it is in either Current or Pending.
func (cs *ConfigState) ContainsServer(serverID int) bool {
```

```
    inCurrent := contains(cs.Current.Servers, serverID)
    if cs.Pending != nil {
        return inCurrent || contains(cs.Pending.Servers, serverID)
    }
    return inCurrent
}

// VotingMembers returns the set of server IDs that are voting members in the current effective configuration.
// For joint consensus, this is the union of Current and Pending servers.
func (cs *ConfigState) VotingMembers() []int {
    if cs.Pending == nil {
        return cs.Current.Servers
    }
    // Union of two slices
    union := make(map[int]bool)
    for _, id := range cs.Current.Servers {
        union[id] = true
    }
    for _, id := range cs.Pending.Servers {
        union[id] = true
    }
    result := make([]int, 0, len(union))
    for id := range union {
        result = append(result, id)
    }
    sort.Ints(result)
    return result
}

// MajoritySize returns the number of votes needed for a majority in the current effective configuration.
func (cs *ConfigState) MajoritySize() int {
    voters := cs.VotingMembers()
    return len(voters)/2 + 1
}

// QuorumMajority checks if the given set of server IDs constitutes a quorum (majority) in the current effective
configuration.
// For joint consensus, it must be a majority in both the old and new configurations.
```

```go
func (cs *ConfigState) QuorumMajority(voters []int) bool {
    if cs.Pending == nil {
        // Simple majority in current configuration.
        return overlapCount(voters, cs.Current.Servers) >= (len(cs.Current.Servers)/2 + 1)
    }
    // Joint consensus: majority in both current and pending.
    majCurr := overlapCount(voters, cs.Current.Servers) >= (len(cs.Current.Servers)/2 + 1)
    majPending := overlapCount(voters, cs.Pending.Servers) >= (len(cs.Pending.Servers)/2 + 1)
    return majCurr && majPending
}

// Helper functions
func contains(slice []int, val int) bool {
    for _, v := range slice {
        if v == val {
            return true
        }
    }
    return false
}

func overlapCount(a, b []int) int {
    set := make(map[int]bool)
    for _, v := range a {
        set[v] = true
    }
    count := 0
    for _, v := range b {
        if set[v] {
            count++
        }
    }
    return count
}

// ConfigCommand is used as the Command field in a LogEntry for configuration changes.
// It wraps a Configuration and marks the type.
```

```go
type ConfigCommand struct {
    Type string        `json:"type"` // "config"
    Config Configuration `json:"config"`
}

// DecodeConfigCommand attempts to decode a LogEntry's Command into a ConfigCommand.
func DecodeConfigCommand(cmd interface{}) (*ConfigCommand, error) {
    data, err := json.Marshal(cmd)
    if err != nil {
        return nil, err
    }
    var configCmd ConfigCommand
    if err := json.Unmarshal(data, &configCmd); err != nil {
        return nil, err
    }
    if configCmd.Type != "config" {
        return nil, fmt.Errorf("not a config command")
    }
    return &configCmd, nil
}
```

D. Core Logic Skeleton Code

We need to extend the `RaftNode` to track configuration state and update the election and replication logic to use joint consensus rules.

First, add configuration fields to `RaftNode`:

```go
// internal/raft/node.go (add to RaftNode struct)
type RaftNode struct {
    mu sync.Mutex
    // ... existing fields ...

    // Configuration state
    configState ConfigState
    // Learners are non-voting members that are catching up.
    learners map[int]bool // key: server ID, value: true if learner

    // ... rest of existing fields ...
}
```

Update the `handleRequestVote` method to respect configuration:

```go
// internal/raft/consensus.go

// handleRequestVote processes a vote request.
// It must now check if the requesting candidate is in the current voting configuration.
func (rn *RaftNode) handleRequestVote(args RequestVoteArgs) RequestVoteReply {
    rn.mu.Lock()
    defer rn.mu.Unlock()

    reply := RequestVoteReply{Term: rn.currentTerm, VoteGranted: false}

    // 1. Reply false if term < currentTerm
    if args.Term < rn.currentTerm {
        return reply
    }

    // 2. If RPC request or response contains term T > currentTerm: set currentTerm = T, convert to follower
    if args.Term > rn.currentTerm {
        rn.becomeFollower(args.Term)
        // Do not return here; continue to evaluate the vote.
    }

    // TODO 1: Check if the candidate is a voting member in our current effective configuration.
    //         Use rn.configState.ContainsServer(args.CandidateID).
    //         If not, return VoteGranted = false (do not grant vote to non-members).

    // TODO 2: Check if we have already voted in this term (rn.votedFor is -1 or equals args.CandidateID).
    //         This logic remains the same as before.

    // TODO 3: Check if candidate's log is at least as up-to-date as ours.
    //         This logic remains the same (compare last log term and index).

    // If all checks pass, grant vote.
    // TODO 4: Update rn.votedFor and persist.

    return reply
}
```

Extend the `startReplicationLoop` and `broadcastAppendEntries` to handle learners and configuration entries:

```go
// internal/raft/consensus.go


// startReplicationLoop runs in a goroutine for the leader to replicate entries to followers and learners.

func (rn *RaftNode) startReplicationLoop() {

    for rn.state == Leader {

        rn.mu.Lock()

        // TODO 1: Determine list of targets: includes all voting members (from configState.VotingMembers())

        //         and all learners (from rn.learners).

        //         For each target, if it's a learner, it does not count toward commit majorities.

        rn.mu.Unlock()



        // TODO 2: For each target, send AppendEntries RPC with appropriate entries.

        //         This logic is similar to before, but note that learners do not have matchIndex/nextIndex tracking?

        //         Actually, we can still track matchIndex for learners to know their progress, but they are not in
matchIndex array (which is indexed by peer ID).

        //         We may need a separate map for learner progress.



        time.Sleep(50 * time.Millisecond) // replication interval

    }

}


// handleAppendEntries must also be updated to reject entries from leaders that are not in the receiver's configuration.

// This is a safety measure to prevent removed servers from affecting the cluster.

func (rn *RaftNode) handleAppendEntries(args AppendEntriesArgs) AppendEntriesReply {

    rn.mu.Lock()

    defer rn.mu.Unlock()



    reply := AppendEntriesReply{Term: rn.currentTerm, Success: false}



    // 1. Reply false if term < currentTerm

    if args.Term < rn.currentTerm {

        return reply

    }



    // 2. If RPC request or response contains term T > currentTerm: set currentTerm = T, convert to follower

    if args.Term > rn.currentTerm {

        rn.becomeFollower(args.Term)
```

```
    }



    // TODO 1: Check if the leader is a voting member in our current effective configuration.

    //         Use rn.configState.ContainsServer(args.LeaderID).

    //         If not, return Success = false (reject entries from non-members).

    //         Note: This check is not in the original Raft paper, but it is a practical safety measure.

    //         Alternatively, we could allow AppendEntries from any server, but then we must ensure that only
configurations we accept are from valid leaders.



    // ... rest of the existing AppendEntries logic (log consistency check, etc.) ...



    return reply

}
```

Implement the configuration change procedure on the leader:

```go
// internal/raft/consensus.go


// ProposeConfigurationChange is called by the client/admin to propose a new configuration.

// It must be called on the leader.

func (rn *RaftNode) ProposeConfigurationChange(newServers []int) error {

    rn.mu.Lock()

    defer rn.mu.Unlock()



    if rn.state != Leader {

        return fmt.Errorf("not leader")

    }



    // TODO 1: Check that no configuration change is already in progress (configState.Pending == nil).

    //         If there is, return an error (only one configuration change at a time).



    // TODO 2: Create the new configuration.

    newConfig := Configuration{Servers: newServers}



    // TODO 3: If we are not in a joint consensus (configState.Pending == nil), then we need to create a joint consensus
entry.

    //         The joint consensus entry includes both old and new servers (union of current and new).

    //         Build the joint configuration (union of configState.Current.Servers and newServers).

    //         Create a ConfigCommand with this joint configuration and append it to the log.

    //         Update configState.Pending to point to the new configuration (newServers) ??

    //         Actually, in joint consensus, the pending configuration is the new configuration, and the current remains
the old.

    //         The intermediate log entry contains the joint configuration (Cold,new).



    // TODO 4: If we are already in a joint consensus (configState.Pending != nil), then we are ready to commit the new
configuration.

    //         This means the joint consensus entry has been committed, and we now append the new configuration entry
(Cnew).

    //         After this new entry is committed, the configuration change is complete.



    // TODO 5: Replicate the configuration entry via broadcastAppendEntries.



    return nil

}
```

Update the commit index advancement to use joint consensus quorums:

```go
// internal/raft/consensus.go


// advanceCommitIndex updates the leader's commit index based on matchIndex.

// This must now consider the configuration state for quorum calculations.

func (rn *RaftNode) advanceCommitIndex() {

    // TODO 1: Gather matchIndex values for all voting members (excluding learners).

    //         Sort them to find the median (N-th largest where N = majority size).



    // TODO 2: For each index candidate, check if it is replicated to a quorum according to the current configuration
state.

    //         Use rn.configState.QuorumMajority(serversThatHaveReplicatedThisIndex).

    //         Note: The quorum must be calculated based on the configuration at that index.

    //         However, for simplicity, we can use the latest configuration (committed) for all indices.

    //         A more accurate implementation would use the configuration at each index (as per the paper).



    // TODO 3: Apply the safety rule: only commit entries from the current term (or older terms if they are safe).

    //         This rule remains unchanged from Milestone 3.



    // TODO 4: Update rn.commitIndex and apply committed entries.

}
```

E. Language-Specific Hints

- Use `encoding/json` to serialize configuration commands within log entries.
- When storing configurations, ensure they are part of the persistent state (along with `currentTerm`, `votedFor`, and log). The `Configuration` struct should be saved to disk when changed.
- For learners, consider adding a separate map in `RaftNode` to track learner progress (`nextIndex` and `matchIndex` for learners). Learners do not vote, so they are not included in majority calculations.
- Use a `sync.Map` or a regular map with a mutex for the `learners` set if concurrency is high.

F. Milestone Checkpoint

After implementing membership changes, verify the following scenario:

1. Start a 3-node cluster (IDs 1,2,3).
2. Propose a configuration change to add servers 4 and 5.
3. Observe that the leader logs a joint consensus entry (Cold,new) containing servers {1,2,3,4,5}.
4. After this entry is committed, observe that the leader then logs a new configuration entry (Cnew) containing servers {1,2,3,4,5} (or maybe just {4,5} if removing old ones—but typically we add).
5. During the transition, verify that no leader can be elected without a majority from both old and new sets (e.g., if server 3 is partitioned, the remaining {1,2} cannot elect a leader because they lack a majority in the new set).
6. After the new configuration is committed, verify that servers 4 and 5 participate in voting, and servers 1,2,3,4,5 form the new cluster.

Run your tests with:

```bash
go test ./internal/raft/... -run TestMembershipChange -v
```

Expected output should show configuration entries being replicated and committed, and the cluster maintaining availability throughout the change.

G. Debugging Tips

Symptom	Likely Cause	How to Diagnose	Fix
Configuration change stalls indefinitely	Leader crashed before committing joint consensus entry	Check logs of all servers for configuration entries; see if any server has a different configuration entry.	Ensure that when a new leader is elected, it continues the configuration change from the last committed configuration.
Removed server continues to start elections	Server not detecting it has been removed	Check if the removed server has committed the configuration entry that excludes it.	Implement logic: when a server sees a committed configuration that doesn't include itself, it stops its election timer and becomes a learner.
New server never becomes voting member	Leader not catching up the learner before promoting	Check the leader's log for the new server's progress; is it being replicated to?	Ensure the leader replicates entries to learners and only promotes when caught up (e.g., within 100 entries of the leader's log).
Split brain during configuration change	Not using joint consensus, or incorrect quorum calculation	Check if two leaders are elected in the same term. Look at the configurations they are using.	Implement joint consensus and verify that `QuorumMajority` requires majorities in both old and new during transition.

9. Interactions and Data Flow

> **Milestone(s):** 2 (Log Replication), 3 (Safety Properties)

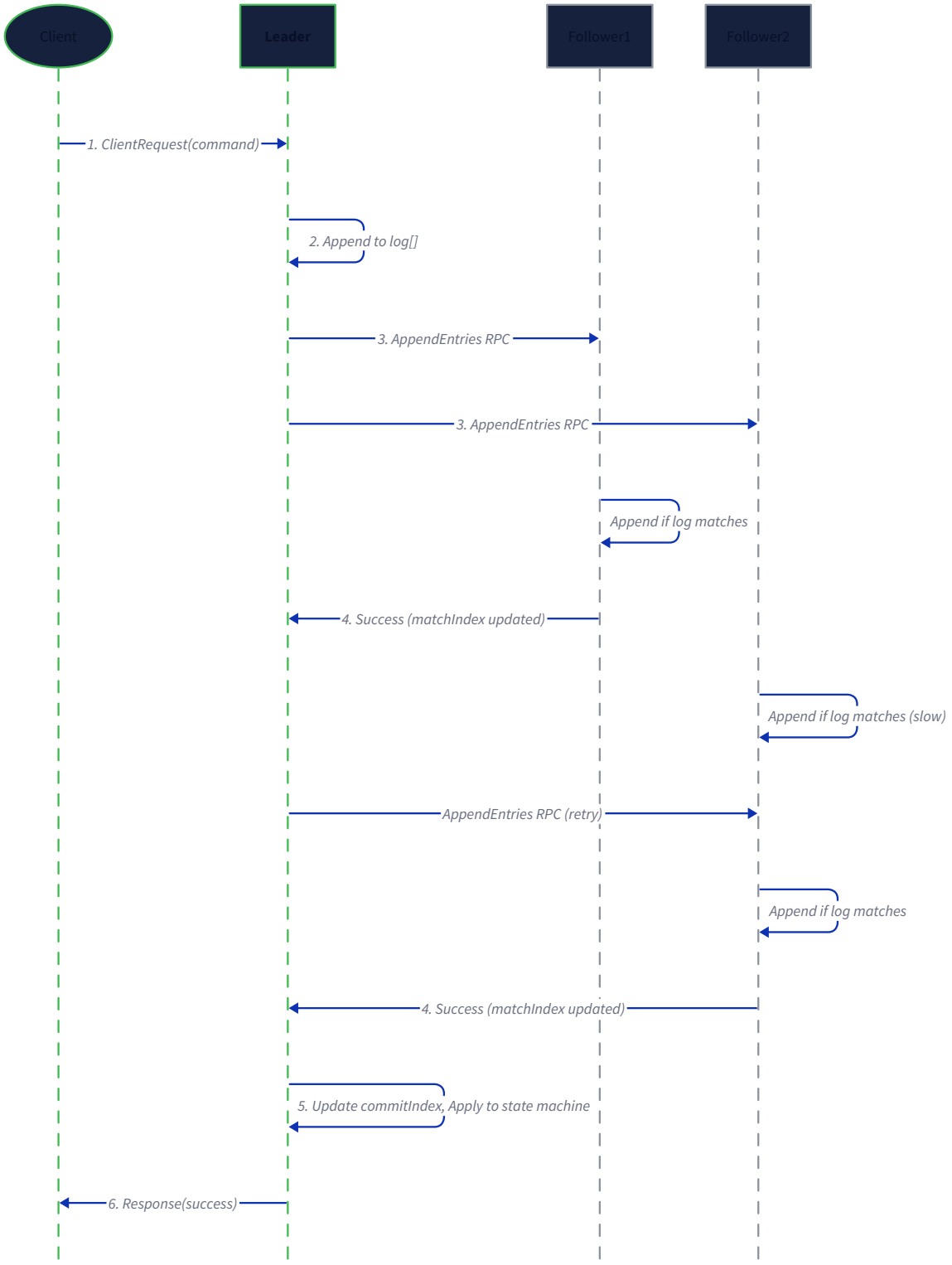
This section traces the journey of a client command from initial proposal through consensus and eventual application on every node. Understanding these interactions reveals Raft's core operational patterns—the normal-case heartbeat of the protocol when no failures occur. We'll examine both the high-level sequence of events and the precise message formats that enable reliable communication between nodes.

9.1 Sequence Diagram: Normal Case Operation

Mental Model: The Military Chain of Command

Imagine a military unit with a clear hierarchy: a commanding officer (the leader) receives orders from headquarters (the client). The officer writes the order in their official logbook, then dispatches runners (RPCs) to all subordinate officers (followers) with a copy of the order. Each subordinate verifies their own logbook matches at the previous entry, appends the new order, and sends back an acknowledgment. Once the commanding officer receives acknowledgments from a majority of subordinates, they mark the order as "executed" in their logbook, carry it out themselves, and notify headquarters of completion. Meanwhile, the runners continuously relay the order to any lagging subordinates until all have it recorded. This strict protocol ensures every officer acts on exactly the same orders in the same sequence, maintaining unit cohesion even if some runners are delayed or the commanding officer changes.

The following sequence diagram illustrates this process for a typical client request:



Walk-Through: A Concrete Example

Let's trace a specific command— `SET user:123 "Alice"` (set key "user:123" to value "Alice")—through a 3-node cluster where Node 2 is the current leader.

1. **Client Request (Step 1):** A client application (e.g., a key-value store client) sends a `ProposeCommand` request to the leader's exposed API endpoint. The request contains the command as an opaque byte array: `SET user:123 "Alice"`. The client may include a unique client ID and sequence number for exactly-once semantics, but that's an application-level concern beyond Raft's core.
2. **Leader Log Append (Step 2):** The leader's `RaftNode` receives the command through its transport layer. It generates a new `LogEntry` with:
   - `Term`: The leader's current term (e.g., 5)
   - `Index`: The next available log index (e.g., 42, since the log currently ends at index 41)
   - `Command`: The serialized `SET` command

   The leader appends this entry to its persistent `LogStore` (using `LogStore.Append`). Crucially, this append happens *before* any replication attempt—the leader cannot propose entries it hasn't first recorded locally. The leader also updates its volatile `nextIndex` and `matchIndex` arrays for all followers, though initially `matchIndex[peer]` for each follower remains at the previous index.
3. **AppendEntries RPC Dispatch (Step 3):** The leader's replication loop (`broadcastAppendEntries`) detects the new log entry. It constructs `AppendEntriesArgs` for each follower:
   - `Term`: 5 (leader's term)
   - `LeaderID`: 2
   - `PrevLogIndex`: 41 (the index immediately before the new entry)
   - `PrevLogTerm`: The term of the entry at index 41 (e.g., 4)
   - `Entries`: A slice containing the single new `LogEntry` at index 42
   - `LeaderCommit`: The leader's current `commitIndex` (e.g., 40)

   The leader sends these RPCs concurrently to all followers (Nodes 1 and 3 in our example). This is not a simple broadcast but targeted replication: each follower receives entries starting from where the leader believes that follower's log ends (`nextIndex[peer]`).
4. **Follower Consistency Check (Step 4):** Each follower receives the RPC and executes `handleAppendEntries`. The follower performs the **consistency check**:
   - Does the follower's log contain an entry at `PrevLogIndex` (41) with term matching `PrevLogTerm` (4)?
   - In the normal case where the follower is up-to-date: Yes. The follower had previously appended entry 41 (from an earlier AppendEntries RPC) with term 4.

   Since the check passes, the follower:
   - Appends the new entry (index 42) to its log
   - Updates its `commitIndex` to `min(LeaderCommit, lastLogIndex)` (so if `LeaderCommit` is 40, the follower's `commitIndex` becomes 40, not 42 yet)
   - Returns `AppendEntriesReply` with `Success: true` and `Term: 5` (matching the leader's term)
5. **Leader Processes Acknowledgments (Step 5):** The leader receives successful replies from followers. For each follower that replied successfully, the leader:
   - Updates `nextIndex[peer]` to `prevLogIndex + len(Entries) + 1` (42 + 1 = 43)
   - Updates `matchIndex[peer]` to `prevLogIndex + len(Entries)` (42)

   After processing replies, the leader checks whether the entry is now **replicated to a majority**. For a 3-node cluster, a majority is 2 nodes (including the leader itself). Since the leader (1) plus at least one follower (1) = 2, the entry at index 42 is now on a majority.
6. **Commit Index Advancement (Step 6):** The leader executes `advanceCommitIndex`. This function applies Raft's **safety rule**:
   - It finds the highest `N` such that `N > commitIndex` and a majority of `matchIndex[peer] >= N`
   - Additionally, for safety (Milestone 3), it requires that the entry at index `N` has `term == currentTerm` (leader's term 5)

   Since entry 42 has term 5 and is on a majority, the leader advances `commitIndex` from 40 to 42.
7. **Leader Applies Committed Entry (Step 7):** The leader's `applyCommittedEntries` routine notices `commitIndex > lastApplied`. It:
   - Iterates from `lastApplied+1` (41) to `commitIndex` (42)
   - For each entry, passes the `Command` to the `StateMachine.Apply` method
   - For index 42, the state machine executes `SET user:123 "Alice"`

- Updates `lastApplied` to 42

The state machine's apply is **deterministic**: given the same sequence of commands, all nodes will reach identical state.

8. **Client Response (Step 8):** Once the leader has applied the command (or in some implementations, once it's committed), it returns a success response to the client. The client can now be confident that the command is durable and will eventually be executed by all nodes, even if the leader fails immediately after responding.
9. **Follower Catch-Up (Steps 9-10):** Followers 1 and 3 haven't yet advanced their own `commitIndex` to include entry 42. They learn about the new commit point in one of two ways:
   - **Next heartbeat**: The leader's subsequent `AppendEntries` RPC (which may be an empty heartbeat) includes `LeaderCommit: 42`
   - **Next replication batch**: If another client command arrives, its `AppendEntries` RPC will carry the updated `LeaderCommit`

   Upon receiving `LeaderCommit: 42`, each follower advances its own `commitIndex` to 42, then applies entries 41-42 (if not already applied) through its own `applyCommittedEntries` routine. All nodes now have identical applied state.

Critical Insight: The Two-Phase Commit Pattern

> Raft's normal operation follows a classic two-phase pattern: (1) **Replication Phase**: The leader disseminates the entry to followers and collects acknowledgments; (2) **Commit Phase**: Once a quorum acknowledges, the leader marks the entry committed and notifies followers. Unlike traditional two-phase commit, Raft has no explicit "prepare" vote—followers accept or reject based solely on log consistency, and the leader decides commitment unilaterally based on quorum. This eliminates coordinator bottleneck while maintaining safety through the quorum intersection property.

Retry Logic for Slow Followers

The sequence diagram shows a retry loop for Follower 2. If a follower is slow, crashes, or experiences network loss, the leader's initial `AppendEntries` RPC may fail or timeout. The leader's replication loop handles this by:

1. **Decrementing `nextIndex`**: When an `AppendEntries` RPC fails (or returns `Success: false`), the leader decrements `nextIndex[peer]` and retries with earlier log entries.
2. **Finding common ground**: This continues until the leader finds a point where its log matches the follower's log (the *consistency check* passes).
3. **Re-sending missing entries**: Once consistency is established, the leader re-sends all entries from that point forward.

This **log reconciliation** process ensures followers eventually catch up, even after extended partitions or crashes.

9.2 Detailed Message Formats

Raft nodes communicate exclusively through two RPC types: `RequestVote` for elections and `AppendEntries` for replication and heartbeats. Each RPC carries precise metadata that enables the protocol's safety guarantees. The following tables detail every field's purpose, semantics, and edge-case handling.

RequestVote RPC (Election Protocol)

The `RequestVote` RPC implements the "campaigning" phase of leader election. A candidate sends this to all peers when its election timeout expires, requesting their vote for leadership in the current term.

Request Structure (`RequestVoteArgs`):

Field	Type	Description & Semantics
`Term`	`int`	**Candidate's current term**. The candidate increments its term when transitioning to Candidate state. This field serves dual purposes: (1) informs recipients of the candidate's view of logical time, (2) if the recipient's term is higher, the candidate will learn of this and revert to Follower.
`CandidateID`	`int`	**Unique identifier of the candidate**. Used by followers to record `votedFor` (persistent state) to ensure they vote at most once per term. Also used in reply addressing.
`LastLogIndex`	`int`	**Index of candidate's last log entry**. Used in the *log comparison* check. Followers grant votes only if the candidate's log is at least as up-to-date as their own. This index represents the highest position in the candidate's log (0 if log is empty).
`LastLogTerm`	`int`	**Term of candidate's last log entry**. The second component of log comparison. Combined with `LastLogIndex`, this allows followers to determine which log is more "up-to-date" using the rule: compare last log terms; if equal, compare last log indices.

Response Structure (`RequestVoteReply`):

Field	Type	Description & Semantics
`Term`	`int`	**Responder's current term**. Used for term updates: if `reply.Term > currentTerm`, the candidate (or any node receiving the reply) must update its term and revert to Follower. This ensures stale candidates learn about newer terms.
`VoteGranted`	`bool`	**Vote decision**. `true` if the voter grants its vote; `false` otherwise. A voter grants its vote if ALL conditions are met: (1) `args.Term >= currentTerm`, (2) `votedFor` is null or equals `args.CandidateID` for this term, (3) candidate's log is at least as up-to-date as voter's log.

RPC Handler Logic (`handleRequestVote`):

The receiver processes the request through these decision steps:

1. **Term validation**: If `args.Term < currentTerm`, reply immediately with `VoteGranted: false` and current `Term`.
2. **Term update**: If `args.Term > currentTerm`, update `currentTerm` to `args.Term`, transition to Follower state, reset `votedFor` (to null), and continue processing.
3. **Vote freshness**: Check if already voted this term (`votedFor` is set). If `votedFor` is null or equals `args.CandidateID`, proceed; otherwise reply `false`.
4. **Log completeness**: Perform log comparison:
   - Compare `args.LastLogTerm` with receiver's last log term
   - If candidate's last log term > receiver's last log term: candidate's log is more up-to-date → grant vote
   - If terms equal: compare `args.LastLogIndex` with receiver's last log index; if candidate's ≥ receiver's → grant vote
   - Otherwise: candidate's log is less up-to-date → deny vote
5. **Record vote**: If granting vote, persist `votedFor = args.CandidateID` (to survive crashes) and reset election timeout (since voting indicates a live candidate).

> **Design Insight**: The log comparison rule ensures **Leader Completeness**—only nodes with all committed entries can become leader. By requiring candidates to have logs at least as up-to-date as a majority, Raft guarantees any elected leader contains all committed entries from previous terms.

AppendEntries RPC (Replication & Heartbeats)

The `AppendEntries` RPC serves three purposes: (1) replicate log entries, (2) serve as heartbeat to maintain leadership, (3) communicate commit index to followers. An empty `Entries` slice constitutes a heartbeat.

Request Structure (`AppendEntriesArgs`):

Field	Type	Description & Semantics
`Term`	`int`	**Leader's current term**. Establishes the sender's authority. If a follower sees a higher term than its own, it updates its term and reverts to Follower. If a leader receives a reply with higher term, it steps down.
`LeaderID`	`int`	**Identifier of the leader**. Allows followers to redirect clients and reset election timeout upon receiving valid RPC from current leader.
`PrevLogIndex`	`int`	**Index immediately preceding new entries**. Used for consistency check. The leader includes the index where the new entries should follow in the follower's log. If the follower's log doesn't have an entry at this index, or has one with mismatching term, the follower rejects the request.
`PrevLogTerm`	`int`	**Term of entry at PrevLogIndex**. Second part of consistency check. The follower compares this term against the term of its entry at `PrevLogIndex` (if any). Mismatch indicates log inconsistency.
`Entries`	`[]LogEntry`	**Log entries to append (empty for heartbeat)**. A slice of `LogEntry` structures to append to the follower's log. If non-empty, the leader guarantees these entries are consecutive starting from `PrevLogIndex+1`. Followers must truncate any existing conflicting entries at or after this position before appending.
`LeaderCommit`	`int`	**Leader's commit index**. The leader informs followers of its `commitIndex`. Followers update their own `commitIndex` to `min(LeaderCommit, lastLogIndex)`. This propagates commit information without extra RPCs.

Response Structure (`AppendEntriesReply`):

Field	Type	Description & Semantics
`Term`	`int`	**Responder's current term**. Used for term updates. If `reply.Term > currentTerm`, leader must update its term and transition to Follower.
`Success`	`bool`	**Result of consistency check**. `true` if `PrevLogIndex` and `PrevLogTerm` matched follower's log and entries were appended (or already present). `false` indicates log inconsistency—the follower's log diverges from the leader's at `PrevLogIndex`.

RPC Handler Logic (`handleAppendEntries`):

The follower processes the request through these steps:

1. **Term validation**: If `args.Term < currentTerm`, reply immediately with `Success: false` and current `Term`.
2. **Term update & leadership discovery**: If `args.Term > currentTerm`, update `currentTerm` to `args.Term`, transition to Follower, reset `votedFor`, and continue. Also, reset election timeout (valid RPC from current leader).
3. **Reset election timeout**: Even if terms equal, receiving a valid RPC from the current leader resets the follower's election timer, preventing unnecessary elections.
4. **Consistency check**: Check if log contains entry at `PrevLogIndex` with term equal to `PrevLogTerm`:
   - **Success case**: Log matches at the previous index. Append any new entries (truncating conflicting ones if necessary), reply `Success: true`.
   - **Failure case**: Log doesn't match (missing entry or term mismatch). Reply `Success: false`. The leader will need to retry with earlier `PrevLogIndex`.
5. **Update commit index**: If `args.LeaderCommit > commitIndex`, set `commitIndex = min(args.LeaderCommit, lastLogIndex)`. This may trigger application of newly committed entries.

Empty Entries as Heartbeats:

When `Entries` slice is empty (length 0), the RPC serves as a **heartbeat**. The leader sends these periodically (typically every 50-150ms, much shorter than election timeout) to:

- Maintain its authority by demonstrating liveness
- Prevent followers from starting elections
- Propagate `LeaderCommit` to followers
- Opportunistically discover if any follower has higher term

Even heartbeats include `PrevLogIndex` and `PrevLogTerm` (typically the leader's last log index/term), which allows followers to confirm they're still consistent with the leader.

Log Truncation on Conflict:

When a follower receives non-empty `Entries` and the consistency check succeeds but the follower already has entries at the same indices, it must handle conflicts:

1. Find the first index where the existing entry differs from the new entry (by comparing terms).
2. Truncate the log from that index onward (`LogStore.Truncate(fromIndex)`).
3. Append the new entries starting at that index.

This ensures the **Log Matching Property**: if two logs have an entry with the same index and term, they are identical in all preceding entries. The leader's log is always considered authoritative.

RPC Timeout and Retry Semantics

Both RPC types share common retry characteristics:

Aspect	Specification	Rationale
Timeout Duration	Typically 1-2x expected RPC round-trip time (e.g., 100-300ms). Must be less than election timeout to avoid false leader changes.	Balances between timely failure detection and avoiding unnecessary retries due to transient network delays.
Retry Limit	No fixed limit; retry indefinitely while leader remains active. For elections, candidates retry until they win, lose, or discover higher term.	Ensures eventual progress despite temporary network issues.
Backoff Strategy	For log replication: immediate retry with decremented `nextIndex`. For elections: random timeout before new election if split vote occurs.	Prevents network congestion while making progress.
Concurrent RPCs	Leader may have multiple outstanding `AppendEntries` RPCs to the same follower for different log positions. Must handle replies out-of-order.	Increases throughput; requires careful `nextIndex` / `matchIndex` updates to avoid regressions.

Message Flow for Configuration Changes (Milestone 4)

For cluster membership changes, the message flow follows the same pattern but with special log entries:

1. **Proposal**: Client calls `ProposeConfigurationChange(newServers []int)` on leader.
2. **Log Entry**: Leader creates a `LogEntry` where `Command` is a serialized `ConfigCommand` with `Type: "JOINT"` and `Config` containing both old and new server sets.
3. **Replication**: Leader replicates this entry via normal `AppendEntries` RPCs, but the **majority calculation** requires quorums from both old and new configurations.
4. **Commit**: Once the joint consensus entry commits, the leader logs a second entry with `Type: "NEW"` containing only the new configuration.
5. **Completion**: When the new configuration entry commits, the cluster operates exclusively with the new servers. Removed nodes stop receiving RPCs; new nodes become full voting members.

The key difference is in the `ConfigState.QuorumMajority` check within `advanceCommitIndex`, which must account for the joint consensus requirement during the transition.

9.3 Interaction Patterns Summary

The following table summarizes the primary interaction patterns in Raft's normal operation:

Pattern	Trigger	Participants	Key Messages	Outcome
Command Replication	Client submits command to leader	Leader, Followers (majority)	`AppendEntries` with entries	Command committed and applied on all nodes
Heartbeat Maintenance	Leader's heartbeat timer expires	Leader, Followers	`AppendEntries` (empty)	Leadership affirmed, follower timeouts reset
Leader Election	Follower election timeout expires	Candidate, All voting members	`RequestVote` requests/replies	New leader elected or split vote retry
Log Reconciliation	Follower log inconsistency detected	Leader, Lagging follower	Series of `AppendEntries` with decrementing `PrevLogIndex`	Follower log brought up-to-date with leader
Configuration Change	Admin requests membership change	Leader, Old & new configuration members	`AppendEntries` with config entries	Cluster safely transitions to new membership
Term Discovery	Node receives RPC with higher term	Any two nodes	Any RPC with `Term > currentTerm`	Receiver updates term, reverts to Follower

These patterns compose to form Raft's complete behavior, with each pattern maintaining specific safety invariants even when composed concurrently.

Implementation Guidance

This section provides concrete implementation starters for the message handling and interaction patterns described above.

Technology Recommendations Table

Component	Simple Option	Advanced Option
RPC Transport	HTTP/1.1 with JSON serialization using Go's `net/http`	gRPC with Protocol Buffers for binary efficiency
Message Serialization	Standard `encoding/json` package	Custom binary format with `encoding/binary` for performance
Concurrent RPC Handling	Goroutine per RPC with `sync.Mutex` for state protection	Worker pool with bounded parallelism and request batching
Timeout Management	`time.Timer` with reset capability	Custom `Timer` struct with jitter and statistical adaptation

Recommended File Structure

Extend the existing project structure with transport and message handling:

```
project-root/
├── cmd/
│   └── raft-node/
│       └── main.go             # Node entry point
├── internal/
│   ├── raft/
│   │   ├── node.go             # RaftNode struct and main loop
│   │   ├── election.go         # Election logic (Milestone 1)
│   │   ├── replication.go      # Replication logic (Milestone 2)
│   │   ├── safety.go           # Safety rules (Milestone 3)
│   │   ├── membership.go       # Membership changes (Milestone 4)
│   │   ├── state.go            # Persistent/volatile state types
│   │   └── messages.go         # RPC struct definitions
│   ├── transport/
│   │   ├── interface.go        # Transport interface
│   │   ├── http.go             # HTTP implementation
│   │   └── mock.go             # Mock for testing
│   ├── logstore/
│   │   ├── interface.go        # LogStore interface
│   │   └── memory.go           # In-memory implementation
│   └── statemachine/
│       ├── interface.go        # StateMachine interface
│       └── kv.go               # Example key-value store
└── pkg/
    └── client/
        └── raft_client.go      # Client library for talking to Raft cluster
```

Complete HTTP Transport Implementation

Here's a complete, ready-to-use HTTP transport layer that handles RPC serialization and delivery:

```go
// internal/transport/http.go

package transport

import (
    "bytes"
    "encoding/json"
    "fmt"
    "net/http"
    "sync"
    "time"
)

// HTTPTransport implements the Transport interface using HTTP/JSON
type HTTPTransport struct {
    mu       sync.RWMutex
    nodeID   int
    addr     string
    server   *http.Server
    clients  map[int]*http.Client  // HTTP clients to other nodes
    handler  func(Message) (interface{}, error)
    timeout  time.Duration
}

// NewHTTPTransport creates a new HTTP transport for the given node
func NewHTTPTransport(nodeID int, addr string, handler func(Message) (interface{}, error)) *HTTPTransport {
    return &HTTPTransport{
        nodeID:  nodeID,
        addr:    addr,
        clients: make(map[int]*http.Client),
        handler: handler,
        timeout: 3 * time.Second, // Default RPC timeout
    }
}

// Start begins listening for incoming RPCs
func (t *HTTPTransport) Start() error {
    mux := http.NewServeMux()
    mux.HandleFunc("/rpc", t.handleRPC)
```

```
    t.server = &http.Server{
        Addr:    t.addr,
        Handler: mux,
    }

    go func() {
        if err := t.server.ListenAndServe(); err != nil && err != http.ErrServerClosed {
            fmt.Printf("HTTP transport failed: %v\n", err)
        }
    }()

    return nil
}

// Send delivers a message to another node
func (t *HTTPTransport) Send(to int, msg Message) (interface{}, error) {
    client, err := t.getClient(to)
    if err != nil {
        return nil, fmt.Errorf("no client for node %d: %w", to, err)
    }

    // Serialize message
    data, err := json.Marshal(msg)
    if err != nil {
        return nil, fmt.Errorf("marshal failed: %w", err)
    }

    // Send HTTP POST
    resp, err := client.Post(fmt.Sprintf("http://node-%d/rpc", to),
        "application/json", bytes.NewReader(data))
    if err != nil {
        return nil, fmt.Errorf("HTTP request failed: %w", err)
    }
    defer resp.Body.Close()
```

```
    // Decode response
    var result map[string]interface{}
    if err := json.NewDecoder(resp.Body).Decode(&result); err != nil {
        return nil, fmt.Errorf("decode response failed: %w", err)
    }

    return result, nil
}

// Stop shuts down the HTTP server
func (t *HTTPTransport) Stop() error {
    if t.server != nil {
        return t.server.Close()
    }
    return nil
}

// handleRPC processes incoming RPC requests
func (t *HTTPTransport) handleRPC(w http.ResponseWriter, r *http.Request) {
    var msg Message
    if err := json.NewDecoder(r.Body).Decode(&msg); err != nil {
        http.Error(w, "bad request", http.StatusBadRequest)
        return
    }

    // Process through handler
    response, err := t.handler(msg)
    if err != nil {
        http.Error(w, err.Error(), http.StatusInternalServerError)
        return
    }

    // Send response
    w.Header().Set("Content-Type", "application/json")
    json.NewEncoder(w).Encode(response)
}
```

```go
// getClient returns or creates an HTTP client for the target node
func (t *HTTPTransport) getClient(nodeID int) (*http.Client, error) {
    t.mu.RLock()
    client, exists := t.clients[nodeID]
    t.mu.RUnlock()

    if exists {
        return client, nil
    }

    // Create new client (in production, you'd know addresses from config)
    t.mu.Lock()
    defer t.mu.Unlock()

    // Double-check after acquiring lock
    if client, exists := t.clients[nodeID]; exists {
        return client, nil
    }

    client = &http.Client{
        Timeout: t.timeout,
    }
    t.clients[nodeID] = client
    return client, nil
}

// SetTimeout updates the RPC timeout duration
func (t *HTTPTransport) SetTimeout(d time.Duration) {
    t.timeout = d
}
```

Core Logic Skeleton: AppendEntries Handler

```go
// internal/raft/replication.go

package raft

import (
    "time"
)

// handleAppendEntries processes an incoming AppendEntries RPC
// This is the core replication logic that followers execute
func (rn *RaftNode) handleAppendEntries(args AppendEntriesArgs) AppendEntriesReply {
    rn.mu.Lock()
    defer rn.mu.Unlock()

    reply := AppendEntriesReply{}

    // TODO 1: Reply false if term < currentTerm (Section 5.1)
    // Compare args.Term with rn.currentTerm
    // If args.Term < rn.currentTerm, set reply.Term = rn.currentTerm, reply.Success = false, return reply

    // TODO 2: If RPC term is greater, update currentTerm and convert to follower (Section 5.1)
    // If args.Term > rn.currentTerm, update rn.currentTerm, set rn.state = Follower, reset rn.votedFor

    // TODO 3: Reset election timeout since we heard from a valid leader (Section 5.2)
    // Call rn.resetElectionTimeout() to prevent unnecessary elections

    // TODO 4: Check log consistency at PrevLogIndex (Section 5.3)
    // Get the term at args.PrevLogIndex using rn.logStore.GetTerm(args.PrevLogIndex)
    // If log doesn't have an entry at PrevLogIndex OR term doesn't match PrevLogTerm:
    //   Set reply.Success = false, reply.Term = rn.currentTerm, return reply

    // TODO 5: Append new entries, truncating if necessary (Section 5.3)
    // If there are existing entries that conflict (same index, different term):
    //   Truncate the log from first conflicting index using rn.logStore.Truncate(conflictIndex)
    // Append new entries using rn.logStore.Append(args.PrevLogIndex, args.PrevLogTerm, args.Entries)

    // TODO 6: Update commit index if LeaderCommit > commitIndex (Section 5.3)
```

```
    // If args.LeaderCommit > rn.commitIndex:
    //   Set rn.commitIndex = min(args.LeaderCommit, lastLogIndex)
    //   Trigger application of newly committed entries (call rn.applyCommittedEntries())

    // TODO 7: Return success
    // Set reply.Success = true, reply.Term = rn.currentTerm
    return reply
}

// broadcastAppendEntries sends AppendEntries RPCs to all followers
// This is called by the leader's replication loop
func (rn *RaftNode) broadcastAppendEntries() {
    rn.mu.RLock()
    if rn.state != Leader {
        rn.mu.RUnlock()
        return
    }

    currentTerm := rn.currentTerm
    leaderID := rn.id
    commitIndex := rn.commitIndex
    rn.mu.RUnlock()

    for _, peerID := range rn.peers {
        if peerID == rn.id {
            continue // Skip self
        }

        go func(peer int) {
            // TODO 1: Prepare AppendEntriesArgs for this follower
            // Get nextIndex for this peer: rn.nextIndex[peer]
            // Get prevLogIndex = nextIndex - 1
            // Get prevLogTerm from log at prevLogIndex (handle case where prevLogIndex = 0)
            // Get entries to send from nextIndex onward using rn.logStore.Slice(nextIndex, lastIndex+1)

            // TODO 2: Send RPC via transport
```

```
            // args := AppendEntriesArgs{Term: currentTerm, LeaderID: leaderID, ...}
            // reply, err := rn.transport.Send(peer, Message{Type: "AppendEntries", Payload: args})

            // TODO 3: Process reply in handleAppendEntriesResult
            // If err != nil (timeout/network error), decrement nextIndex and retry later
            // If reply.Success == true: update nextIndex and matchIndex for this peer
            // If reply.Success == false and reply.Term > currentTerm: convert to follower
            // If reply.Success == false but terms equal: decrement nextIndex and retry
        }(peerID)
    }
}
```

Language-Specific Hints for Go

1. **Concurrency Safety**: Always acquire `rn.mu.Lock()` before accessing any `RaftNode` fields that might be accessed concurrently (which is most fields). Use `defer rn.mu.Unlock()` for safety. For read-only operations, use `rn.mu.RLock()` / `rn.mu.RUnlock()`.
2. **Timer Management**: Use the provided `Timer` struct from earlier sections. Never call `time.Timer.Reset()` without first stopping or draining the timer channel to avoid race conditions.
3. **JSON Serialization**: Use struct tags for JSON marshaling: `type RequestVoteArgs struct { Term int `json:"term"` }`. This ensures field names match Raft paper conventions when serialized.
4. **Goroutine Lifecycle**: Always use `go func() { ... }()` for RPC handling and replication loops. Keep track of goroutines and ensure they exit when the node stops (use `context.Context` for cancellation).
5. **Slice Operations**: When sending log entries in `AppendEntriesArgs`, create a copy of the slice: `entries := make([]LogEntry, len(entriesToSend)); copy(entries, entriesToSend)`. This prevents data races if the log is modified while the RPC is in flight.

Milestone Checkpoint for Interaction Flow

To verify your implementation handles the normal case correctly:

```bash
# Start a 3-node cluster in separate terminals
go run cmd/raft-node/main.go --id=1 --peers=1,2,3 --port=8080
go run cmd/raft-node/main.go --id=2 --peers=1,2,3 --port=8081
go run cmd/raft-node/main.go --id=3 --peers=1,2,3 --port=8082

# Use the test client to send commands
go run cmd/test-client/main.go --leader=localhost:8080 set key1 value1
```

Expected Behavior:

1. One node becomes leader (check logs for "became leader in term X")
2. Client command is accepted by leader
3. Leader replicates entry to followers (logs show "AppendEntries to node Y")
4. Entry commits (logs show "commit index advanced to Z")
5. All nodes apply the command (logs show "applied command at index Z")
6. Client receives success response

Debugging Checklist:

- If no leader emerges: Check election timeout randomization and `RequestVote` RPC handling
- If leader doesn't accept commands: Verify `state == Leader` and log append works
- If followers reject AppendEntries: Check `PrevLogIndex` / `PrevLogTerm` consistency
- If commit index doesn't advance: Verify majority calculation and `matchIndex` updates
- If entries aren't applied: Check `commitIndex > lastApplied` condition

Common Bug Symptoms and Fixes

Symptom	Likely Cause	Diagnosis	Fix
Commands hang forever	Leader not replicating to majority	Check if followers are responding to AppendEntries; check `nextIndex` values	Ensure leader retries failed RPCs with decremented `nextIndex`
Split votes recurring	Election timeouts not randomized enough	Check timer range; nodes starting simultaneously	Increase randomization range (150-300ms typical)
Logs diverge after recovery	Inconsistent `PrevLogTerm` handling	Compare follower's log term at `PrevLogIndex`	Ensure `GetTerm(index)` returns 0 for index 0 (before log starts)
Commit index lags behind	Leader not updating `matchIndex` correctly	Check `handleAppendEntriesResult` logic	Update `matchIndex[peer] = prevLogIndex + len(entries)` on success
Double application of entries	`lastApplied` not updated atomically	Check `applyCommittedEntries` race conditions	Apply entries under mutex protection, update `lastApplied` immediately

10. Error Handling and Edge Cases

> **Milestone(s):** 1 (Leader Election), 2 (Log Replication), 3 (Safety Properties), 4 (Cluster Membership Changes)

Robust error handling is what distinguishes a correct distributed system from a theoretical exercise. Raft's design elegantly transforms various failure modes—server crashes, network partitions, message loss—into well-defined scenarios that the protocol's normal mechanisms can handle. This section catalogs these failure modes, explains how they are detected, and details the recovery procedures that ensure the system maintains its safety guarantees while striving for availability.

10.1 Failure Modes and Detection

Mental Model: The Fault-Tolerant Orchestra Think of a Raft cluster as an orchestra that must keep perfect time despite unpredictable musicians. Some players might doze off (crashes), others might be playing in a soundproof booth (network partitions), and a few might occasionally play wrong notes (transient message corruption). The conductor (leader) uses regular check-ins (heartbeats) to detect who's unresponsive and has a clear protocol for handing the baton (leadership) to another musician when they fail to check in. Crucially, the orchestra has a rule: only the musician with the most complete and up-to-date score (log) can become the new conductor. This mental model helps visualize how Raft turns the chaotic problem of distributed failures into a manageable set of detectable conditions.

Raft is designed for the **crash-stop** failure model, where servers fail by stopping and later may restart, but do not exhibit arbitrary (Byzantine) behavior. The protocol also handles **network partitions** and **message loss**, which are manifestations of the same underlying issue: temporary inability to communicate.

The table below catalogs the primary failure modes, their symptoms, and Raft's detection mechanisms:

Failure Mode	Description	Detection Mechanism in Raft	Timeout/Retry Strategy
Leader Crash	The current leader process terminates abruptly (hardware failure, `kill -9`).	Followers have an `electionTimeout` timer that resets on each valid `AppendEntries` RPC (heartbeat) from the leader. No heartbeat → timer expires.	Election timeout (randomized 150-300ms typical). Followers transition to `Candidate` and start new election.
Follower Crash	A non-leader server stops responding.	Leader detects via failed `AppendEntries` RPCs (network timeout or connection refused). Leader continues retrying indefinitely.	Leader's RPC send retries (with exponential backoff or fixed interval). Does not affect liveness unless majority crashes.
Network Partition	The network splits, isolating some servers from others. 	Partitioned leader cannot reach majority → its `AppendEntries` RPCs fail. Partitioned followers' election timers expire (no heartbeats).	Both sides run independent elections. Only partition with majority can elect a leader and commit entries.
Message Loss	A single RPC (vote request, log entry) is dropped by the network.	Sender receives no reply within RPC timeout. For leaders, missing `AppendEntries` reply; for candidates, missing `RequestVote` reply.	Sender retries the RPC (idempotent operations). Random election timeouts prevent repeated split votes.
Network Delay / Slow Server	Messages arrive but with high latency, or a server is temporarily overloaded (GC pause, I/O spike).	Receivers may process messages with stale terms; senders observe delayed replies that may arrive after timeout.	Timeouts must be longer than normal RPC round-trip. `AppendEntries` consistency check ensures correctness despite delayed messages.
Restarted Server	A crashed server reboots with persistent state intact (or empty if fresh).	On startup, it has stale term and log. It will receive RPCs with higher term, causing it to update term and revert to `Follower`.	Bootstraps as `Follower`, starts election timer. Catches up via leader's `AppendEntries` consistency check and log replication.

Architecture Decision Record: Handling Network Partitions via CP System Design

> **Decision: Prioritize Consistency over Availability during Network Partitions (CP system)**
>
> - **Context**: According to the CAP theorem, during a network partition, a distributed system must choose between Consistency (C) and Availability (A). Raft is designed for systems where correctness is paramount (e.g., configuration management, distributed locks).
> - **Options Considered**:
>   1. **Always maintain availability**: Allow both sides of a partition to continue processing requests (AP system like Dynamo). This leads to split-brain and data divergence.
>   2. **Prioritize consistency, halt minority side**: Ensure only the partition with a majority of servers (quorum) can make progress, halting the minority side (CP system).
>   3. **Hybrid approach**: Use techniques like leader leases or fencing tokens to allow reads on the minority side with stale data, but writes require majority.
> - **Decision**: Raft adopts option 2: during a partition, only the majority side can elect a leader and commit new log entries. The minority side's candidates will continuously increment their terms but cannot win elections (cannot get majority).
> - **Rationale**: This guarantees **safety** (no conflicting commits) at the expense of **availability** for the minority partition. Many real-world applications using Raft (etcd, Consul) accept this trade-off because they prioritize strong consistency. The minority partition remains unavailable until the partition heals, at which point nodes reconcile and catch up.
> - **Consequences**: During a partition, client requests routed to the minority side will fail or timeout. Applications must handle this gracefully (retry, failover). The design simplifies the protocol significantly compared to handling concurrent leaders.

The following table compares the failure detection strategies:

Detection Component	Implementation Mechanism	Tuning Parameter	Rationale for Design
Election Timeout	`Timer` with random duration between `baseTimeout` and `baseTimeout + rangeTimeout`. Reset on valid RPCs from current leader or granting vote to candidate.	`baseTimeout = 150ms`, `rangeTimeout = 150ms` (typical).	Randomization prevents split votes; timeout duration balances recovery speed vs. spurious elections under load.
Heartbeat Interval	Leader periodically sends empty `AppendEntries` RPCs (no log entries) to all followers.	Fixed interval, e.g., `heartbeatInterval = 50ms` (<< election timeout).	Frequent heartbeats suppress follower elections; interval must allow timely detection but not overwhelm network.
RPC Timeout	When sending `RequestVote` or `AppendEntries`, caller waits for reply with a deadline.	`rpcTimeout = 2 * avgRTT` (e.g., 100ms). Should be < election timeout.	Allows quick retry on loss; short enough to not block election progression.
Log Reconciliation Retry	When `AppendEntries` fails due to log inconsistency, leader decrements `nextIndex` for that follower and retries.	No explicit timeout; retries on next heartbeat or immediately.	Eventually finds matching point; exponential backoff optional to avoid network storm.

Common Pitfalls in Failure Detection

⚠️ Pitfall: Using Fixed Election Timeouts

- **Description**: Setting the same election timeout for all servers (e.g., exactly 200ms for each).
- **Why It's Wrong**: If multiple followers' timeouts expire simultaneously (common after leader crash or network partition), they all become candidates simultaneously, causing split votes. No candidate may achieve majority, leading to repeated elections and extended unavailability.
- **Fix**: Implement **randomized timeouts** as specified in the Raft paper. Each node's `resetElectionTimeout()` should pick a random duration within a configured range (e.g., 150-300ms). This ensures statistical dispersion.

⚠️ Pitfall: Not Resetting Election Timer on Appropriate Events

- **Description**: The election timer is reset only when receiving an `AppendEntries` RPC from a valid leader, but not when granting a vote to a candidate.
- **Why It's Wrong**: Consider a partitioned network where two followers can communicate but cannot reach the leader. One becomes candidate, requests vote from the other, which grants it. If the voter doesn't reset its timer, it will soon become a candidate itself, causing unnecessary contention and term inflation.
- **Fix**: Reset the election timer on two events: 1) receiving an `AppendEntries` RPC from a leader with term >= current term, and 2) granting a vote in `handleRequestVote`. This ensures followers remain followers while they believe a valid leader or candidate is active.

⚠️ Pitfall: Ignoring Stale RPC Replies

- **Description**: Processing an RPC reply that arrives very late, after the node has transitioned to a new term or state.
- **Why It's Wrong**: A delayed `RequestVoteReply` from an earlier term could incorrectly affect a current election, or a stale `AppendEntriesReply` could cause a leader to update `matchIndex` incorrectly.
- **Fix**: **Always check the term in the reply against current term before processing**. In the candidate's vote tallying, discard replies with `Term < currentTerm`. In the leader's `handleAppendEntriesResult`, ignore replies where `args.Term != currentTerm` (the leader may have stepped down or updated term since sending). This is crucial for safety.

10.2 Recovery and State Reconciliation

Mental Model: The Archaeological Dig Site Imagine a team of archaeologists (Raft followers) excavating a site, each maintaining their own field notes (log). After a sandstorm (network partition or crash), some archaeologists have been working separately, and their notes may have diverged. When they reunite, they don't throw away all notes; instead, they perform a **consistency check**: they compare the last common verified layer (prevLogIndex/prevLogTerm) and then overwrite any inconsistent newer notes with the authoritative version from the team leader. This ensures everyone has an identical record of the dig, even if some temporarily worked on incorrect assumptions.

Recovery in Raft involves bringing a node back into sync with the cluster after a crash, restart, or network partition. The protocol's **strong consistency guarantees** enable safe reconciliation without complex conflict resolution.

10.2.1 Crash Recovery and Persistent State

When a `RaftNode` crashes and restarts, it reloads its **Persistent State** (`currentTerm`, `votedFor`, and the log entries) from stable storage. This persistent state is crucial for safety: without it, a restarted node could vote for a different candidate in the same term (violating **Election Safety**) or forget committed entries (violating **Leader Completeness**).

The recovery procedure on startup:

1. **Load Persistent State**: Read `currentTerm`, `votedFor`, and the entire log from disk (or a persistent `MemoryStore` simulation).
2. **Initialize Volatile State**: Reset `commitIndex = 0`, `lastApplied = 0`. For leaders, `nextIndex` and `matchIndex` would be initialized after election.
3. **Start as Follower**: Set `state = Follower`. The node does not know if a leader exists, so it starts its election timer.
4. **Begin Normal Operation**: Enter the main event loop (`run()`). The node will either:
   - Receive heartbeats from an existing leader (if one exists) and reset its timer.
   - Time out and start an election if no leader is detected.

> **Key Insight**: The persistent `votedFor` field prevents a restarted node from voting twice in the same term. Imagine a node votes for candidate A, crashes, restarts, and votes for candidate B in the same term. If both A and B could collect a majority, we could have two leaders in the same term. Persisting `votedFor` eliminates this risk.

10.2.2 Log Reconciliation via AppendEntries Consistency Check

The core mechanism for bringing a lagging or divergent follower up-to-date is the **consistency check** in the `AppendEntries` RPC. The leader includes `PrevLogIndex` and `PrevLogTerm` in each RPC, and the follower verifies that its own log contains an entry at `PrevLogIndex` with term `PrevLogTerm`. If not, it rejects the RPC (`Success = false`).

The leader's response to rejection is a **log backtracking algorithm**:

1. **Leader decrements `nextIndex`**: Upon rejection, the leader decrements `nextIndex[peerID]` by one (or more optimistically) and retries the `AppendEntries` RPC with the new previous index.
2. **Repeat until match**: Eventually, the leader will either find a matching index (where terms agree) or reach the beginning of the log (index 0, term 0, which always matches).
3. **Overwrite inconsistent entries**: Once a match is found, the leader sends all subsequent entries from its log, and the follower **truncates its log from that point onward** and appends the leader's entries.

This process guarantees the **Log Matching Property**: if two logs have an entry with the same index and term, they are identical in all preceding entries.

Walk-through Example: Follower Recovery after Partition Consider a 5-node cluster (S1-S5) with S1 as leader in term 3. A network partition isolates S5 from the majority (S1-S4). During the partition:

- Majority side (S1-S4) commits entries with indices 10-20 (term 3).
- S5, unable to receive heartbeats, times out and becomes candidate. It increments term to 4 (and higher) but cannot win election (no majority). It may append a no-op entry from its own candidacy at index 10 (term 4).

When the partition heals:

- S1 (still leader in term 3) sends `AppendEntries` to S5 with `PrevLogIndex=20`, `PrevLogTerm=3`.
- S5's log has only up to index 10, and at index 10, its term is $4 \neq 3$. It rejects.
- S1 decrements `nextIndex[S5]` repeatedly. It may try index 9 (term 3 in leader's log, but S5 has no entry at index 9). Reject.
- Eventually, S1 tries `PrevLogIndex=5` (term 2). Suppose S5's log matches at index 5 (term 2). Success! S1 then sends entries from index 6 onward (terms 2,3), overwriting S5's divergent entries at indices 6-10.
- S5's log is now identical to the leader's up to index 20, and it can then commit and apply entries.

10.2.3 Term Reconciliation and Leader Step-Down

A critical recovery scenario occurs when a partitioned node with a higher term re-joins the cluster. Because Raft requires leaders to have the highest known term, this triggers a **leader step-down**.

Procedure: Discovering Higher Term

1. Any node (follower, candidate, or leader) receiving an RPC with `Term > currentTerm` updates its `currentTerm` to the new term and **reverts to `Follower` state**.
2. It resets `votedFor = nil` (or -1) to allow voting in the new term.
3. If the node was a leader, it stops accepting client requests, stops its replication loops, and steps down.

This ensures that a partitioned leader with a stale term cannot continue operating when a newer term exists elsewhere in the cluster.

Walk-through Example: Partitioned Leader Reintegration Continuing the previous example, after partition heals, S5 (term 4) receives an `AppendEntries` from S1 (term 3). Since 4 > 3, S5 rejects the RPC and includes its higher term in the reply (`Reply.Term = 4`). When S1 receives this reply, it sees `Term = 4 > currentTerm (3)`, so it updates its term to 4 and becomes a follower. The cluster will then hold a new election (term 4), and the most up-to-date node (likely S1, which has the longer log) will become leader again.

10.2.4 Handling Configuration Changes during Partitions

Membership changes (Milestone 4) introduce additional edge cases. The **joint consensus** protocol is specifically designed to prevent **disjoint majorities**—a scenario where two different configurations each have a majority that can elect a leader, leading to split-brain.

Recovery from Failed Configuration Change If a leader crashes during a configuration change (while the joint configuration `C_old,new` is uncommitted), the new leader must determine whether to continue the change or revert. Raft's rule: **configuration entries are committed like any other log entry**. Therefore, a new leader will either have the configuration entry in its log (and continue replicating it) or not (and ignore it). The safety property ensures that only servers with the configuration entry can count it toward their majorities.

Isolation of Removed Servers When a server is removed from the configuration via a committed configuration entry, it is no longer part of the cluster. However, if it is partitioned away and never learns of its removal, it may continue to start elections. To prevent this, removed servers should, upon receiving any RPC with a newer configuration, **shut down their Raft instance** or transition to a non-voting learner state. In our educational implementation, we can log a warning and ignore further participation.

Implementation Guidance

This subsection provides concrete code structures and skeletons to implement the error handling and recovery mechanisms described above.

A. Technology Recommendations

Component	Simple Option (Learning)	Advanced Option (Production)
Failure Detection	Go's `time.Timer` with random timeouts; simple RPC timeout via `context.WithTimeout`	Dedicated ticker goroutine with configurable intervals; adaptive timeouts based on observed latency
Persistent State	In-memory struct with periodic write to a file (JSON) for simulation	Efficient binary format (like Protobuf) with `fsync` on every update; write-ahead log (WAL)
Log Reconciliation	Linear decrement of `nextIndex` on failure	Optimistic binary search or skip-based backtracking to find matching index faster

B. Recommended File/Module Structure

Add files for persistence and recovery logic:

```
project-root/
  internal/raft/
    node.go               # RaftNode main struct and run loop
    election.go           # Election logic, timeout handling
    replication.go        # Log replication, consistency check
    persistence.go        # Persistent state save/load
    recovery.go           # Startup recovery, state reconciliation
    transport.go          # HTTPTransport and RPC handling
  internal/store/
    memory_store.go       # MemoryStore (implements LogStore)
    persistent_store.go   # File-backed store (optional)
  cmd/server/
    main.go               # Entry point, start Raft nodes
```

C. Infrastructure Starter Code

Complete Timer with Random Timeout (for election and heartbeat):

```go
// internal/raft/timer.go

package raft


import (

    "math/rand"

    "sync"

    "time"

)


// Timer is a thread-safe timer that generates random timeouts within a range.

type Timer struct {

    mu           sync.Mutex

    duration     time.Duration

    timer        *time.Timer

    resetChan    chan struct{}

    timeoutChan  chan struct{}

    stopped      bool

    baseTimeout  time.Duration

    rangeTimeout time.Duration

}


// NewTimer creates a new Timer with base and range durations.

func NewTimer(base, rangeDur time.Duration) *Timer {

    t := &Timer{

        baseTimeout:  base,

        rangeTimeout: rangeDur,

        resetChan:    make(chan struct{}, 1),

        timeoutChan:  make(chan struct{}, 1),

    }

    t.reset()

    go t.run()

    return t

}


// run monitors the timer and reset signals.

func (t *Timer) run() {

    for {

        select {
```

```
        case <-t.timer.C:
            t.mu.Lock()
            if !t.stopped {
                select {
                case t.timeoutChan <- struct{}{}:
                default:
                }
            }
            t.mu.Unlock()
        case <-t.resetChan:
            // Timer was reset; continue loop
        }
    }
}

// Reset generates a new random timeout and resets the internal timer.
func (t *Timer) Reset() {
    t.mu.Lock()
    defer t.mu.Unlock()
    if t.stopped {
        return
    }
    // Pick random duration within [base, base+range]
    randRange := time.Duration(rand.Int63n(int64(t.rangeTimeout)))
    t.duration = t.baseTimeout + randRange
    if t.timer == nil {
        t.timer = time.NewTimer(t.duration)
    } else {
        // Stop the old timer and drain channel if needed
        if !t.timer.Stop() {
            select {
            case <-t.timer.C:
            default:
            }
        }
        t.timer.Reset(t.duration)
```

```
    }
    // Non-blocking send to resetChan to signal reset
    select {
    case t.resetChan <- struct{}{}:
    default:
    }
}

// TimeoutChan returns the channel that receives a value when timer expires.
func (t *Timer) TimeoutChan() <-chan struct{} {
    return t.timeoutChan
}

// Stop halts the timer.
func (t *Timer) Stop() {
    t.mu.Lock()
    defer t.mu.Unlock()
    t.stopped = true
    if t.timer != nil {
        t.timer.Stop()
    }
}
```

Simple Persistent State Saver/Loader (file-based simulation):

```go
// internal/raft/persistence.go

package raft


import (

    "encoding/json"

    "io/ioutil"

    "os"

    "path/filepath"

    "sync"

)


// PersistentState represents the data that must survive crashes.

type PersistentState struct {

    CurrentTerm int        `json:"currentTerm"`

    VotedFor    *int       `json:"votedFor"` // nil means not voted

    Log         []LogEntry `json:"log"`

}


// save persists the state to a file (simulation for learning).

func (n *RaftNode) savePersistentState() error {

    n.mu.Lock()

    defer n.mu.Unlock()

    state := PersistentState{

        CurrentTerm: n.currentTerm,

        VotedFor:    n.votedFor,

        Log:         n.log,

    }

    data, err := json.Marshal(state)

    if err != nil {

        return err

    }

    filename := filepath.Join("raft_data", fmt.Sprintf("node%d.json", n.id))

    // In production, you would fsync and write atomically.

    return ioutil.WriteFile(filename, data, 0644)

}


// loadPersistentState loads the state from file, if it exists.

func (n *RaftNode) loadPersistentState() error {
```

```
    filename := filepath.Join("raft_data", fmt.Sprintf("node%d.json", n.id))
    if _, err := os.Stat(filename); os.IsNotExist(err) {
        // No saved state; start fresh
        return nil
    }
    data, err := ioutil.ReadFile(filename)
    if err != nil {
        return err
    }
    var state PersistentState
    if err := json.Unmarshal(data, &state); err != nil {
        return err
    }
    n.mu.Lock()
    defer n.mu.Unlock()
    n.currentTerm = state.CurrentTerm
    n.votedFor = state.VotedFor
    n.log = state.Log
    return nil
}
```

D. Core Logic Skeleton Code

Node Startup Recovery Routine:

```go
// internal/raft/recovery.go

package raft


// recoverNode loads persistent state and initializes volatile state after a crash/restart.

func (n *RaftNode) recoverNode() error {

    // TODO 1: Load persistent state from disk (or simulated file) using loadPersistentState()

    // TODO 2: If load fails (e.g., file corrupt), you may choose to start with empty term and log

    // TODO 3: Initialize volatile state: commitIndex = 0, lastApplied = 0

    // TODO 4: If the log is not empty, set lastApplied to the last applied index from previous run (requires persisting lastApplied in production)

    // TODO 5: Set state = Follower (since we don't know current leader)

    // TODO 6: Reset election timer (call resetElectionTimeout)

    // TODO 7: Start the main run loop in a goroutine

    return nil

}
```

Handling Stale RPC Replies in Leader Replication:

```go
// internal/raft/replication.go

package raft


// handleAppendEntriesResult processes a reply from a follower.

func (n *RaftNode) handleAppendEntriesResult(peerID int, args AppendEntriesArgs, reply AppendEntriesReply) {

    n.mu.Lock()

    defer n.mu.Unlock()


    // TODO 1: Safety check: if reply.Term > n.currentTerm, update currentTerm, revert to Follower, and return

    // TODO 2: Ignore reply if we are no longer leader (state != Leader) or if the term of the args doesn't match currentTerm (stale reply)

    // TODO 3: If success, update nextIndex and matchIndex for this peer

    // TODO 4: If failure due to log inconsistency, decrement nextIndex[peerID] and retry (consider optimizing with binary search)

    // TODO 5: After updating matchIndex, call advanceCommitIndex() to possibly commit new entries

}
```

Step-Down Logic on Discovering Higher Term:

```go
// internal/raft/node.go

package raft


// checkTermAndStepDown updates term and steps down if a higher term is seen.

// Call this whenever receiving an RPC with term > currentTerm or an RPC reply with term > currentTerm.

func (n *RaftNode) checkTermAndStepDown(term int) {

    // TODO 1: Lock the mutex (caller may already hold lock, be careful to avoid deadlock)

    // TODO 2: If term > n.currentTerm, set n.currentTerm = term and n.votedFor = nil

    // TODO 3: If currently Leader or Candidate, transition to Follower state

    // TODO 4: Reset election timer (since we are now follower)

    // TODO 5: Persist state (call savePersistentState) because currentTerm and votedFor changed

}
```

E. Language-Specific Hints

- **Timeouts and Timers**: Use `time.NewTimer` and `time.Reset` carefully. Always stop the timer and drain the channel if needed before resetting to avoid stale triggers. Our `Timer` wrapper handles this complexity.
- **RPC Timeouts**: Use `context.WithTimeout` when making HTTP requests. Set a reasonable timeout (e.g., 100ms) that is less than the election timeout.
- **Persistent State**: In a real system, you must `fsync` after writing persistent state. For simulation, writing to a file is sufficient. Consider using a write-ahead log for better performance.
- **Concurrency**: Use `sync.RWMutex` for the `RaftNode` mutex. Acquire read lock for read-only operations (like generating RPC args) and write lock for state modifications.

F. Milestone Checkpoint for Error Handling

After implementing error handling and recovery, verify with the following test scenario:

Test: Leader Crash and Recovery

1. Start a 3-node cluster.
2. Let a leader be elected.
3. Send a client command (e.g., `SET x=5`) to the leader; ensure it's committed and applied on all nodes.
4. Kill the leader process (Ctrl-C or `kill`).
5. Observe: The remaining two nodes should elect a new leader within election timeout (random, so may take a few cycles).
6. Send another command to the new leader; it should be committed and applied.
7. Restart the crashed leader (it should load its persistent state).
8. Observe: The restarted leader should become a follower, catch up on missed log entries via `AppendEntries`, and apply any committed entries it missed.

Expected Log Output: Look for messages like "stepping down to follower due to higher term", "resetting election timeout", "AppendEntries rejected, decrementing nextIndex", "applying committed entry at index X".

Commands to Run: Use your test harness or run the nodes manually, sending commands via a simple client.

G. Debugging Tips for Error Scenarios

Symptom	Likely Cause	How to Diagnose	Fix
Election timeouts never trigger	Election timer not started, or reset too frequently	Log when timer is reset and when it fires. Check if heartbeats are arriving too frequently.	Ensure timer is started in `runFollower`. Reset only on valid heartbeats, not on every RPC.
Split votes happen repeatedly	Election timeouts not randomized enough or nodes start simultaneously	Check the random range; ensure each node picks a distinct seed.	Increase the random range (e.g., 150-300ms). Use proper random seeding (`rand.Seed(time.Now().UnixNano())`).
Follower logs diverge and never converge	Log reconciliation not retrying after failure, or `nextIndex` not decremented	Log `AppendEntries` rejections and the `nextIndex` values.	Implement the backtracking loop in leader's replication logic. Ensure leader retries after failure.
Restarted node votes in same term it voted before crash	`votedFor` not persisted before sending vote reply	Log when `votedFor` is saved. Check persistence call sequence.	Persist `votedFor` (and `currentTerm`) **before** replying to `RequestVote`.
Leader doesn't step down when receiving higher term	Missing term check in RPC handler or reply processing	Log terms in all RPCs and replies. Compare with current term.	Call `checkTermAndStepDown` in every RPC handler and reply handler when a higher term is seen.
Commit index doesn't advance during partition	Leader cannot reach majority; safety rule prevents committing old term entries	Check if entries are from current term. Monitor `matchIndex` values.	This is normal behavior during partition (CP system). Ensure leader only commits entries from its own term when a majority exists.

11. Testing Strategy

> **Milestone(s):** 1 (Leader Election), 2 (Log Replication), 3 (Safety Properties), 4 (Cluster Membership Changes)

Validating a distributed consensus algorithm is fundamentally different from testing a typical application. The correctness of Raft depends on complex interactions between nodes, timing, and network conditions that are difficult to reproduce deterministically. This section provides a comprehensive testing strategy that combines milestone-by-milestone verification with property-based testing to build confidence in the implementation's correctness. We approach testing as a multi-layered process: first verifying each component in isolation, then validating the system's behavior under controlled scenarios, and finally stress-testing it with randomized faults to uncover subtle bugs.

The Raft paper itself provides a formal proof of safety, but as implementers, we must translate those proofs into concrete testable assertions. Our testing philosophy follows three principles: (1) **Test invariants, not just scenarios**—continuously verify properties like "no two leaders in the same term"; (2) **Embrace controlled chaos**—introduce network partitions, message loss, and node crashes to ensure the system maintains safety; and (3) **Make state visible**—build extensive logging and inspection tools to understand the system's behavior when tests fail.

11.1 Milestone Verification Checkpoints

Each milestone in the project builds upon the previous one, so testing should follow the same incremental approach. The following tables provide concrete verification procedures for each milestone, including specific scenarios to simulate, expected behaviors to observe, and commands to execute. These checkpoints assume you've implemented basic logging (using `fmt.Printf` or a structured logger) that outputs key events like state transitions, RPCs sent/received, and term changes.

Milestone 1: Leader Election Verification

The goal for this milestone is to verify that a cluster can reliably elect a single leader, handle election timeouts, and prevent split votes through randomized timeouts.

Test Scenario	Expected Behavior	Verification Steps	Pass/Fail Criteria
Single-node startup	The single node should immediately become leader (no other nodes to contest).	1. Start a single `RaftNode` with ID 1 and empty peer list. 2. Wait for election timeout period (150-300ms). 3. Observe logs.	Node transitions to `Candidate`, increments term, votes for itself, becomes `Leader`. No split vote occurs.
Three-node cluster, all healthy	Exactly one node should be elected leader within a few election cycles.	1. Start three nodes (IDs 1, 2, 3) that can communicate. 2. Wait 1-2 seconds. 3. Observe logs from all nodes.	One node logs `became leader`, others remain `Follower`. All nodes agree on the same `currentTerm`.
Leader failure and re-election	After leader fails, a new leader should be elected.	1. Start three healthy nodes, let leader be elected (note which node). 2. Stop the leader process. 3. Wait 500-1000ms. 4. Observe remaining nodes' logs.	Remaining nodes detect leader absence via election timeout, one becomes new leader. New term number is higher than previous.
Randomized timeout split vote resolution	Split votes (multiple candidates) should resolve after random timeouts.	1. Start three nodes with identical (non-random) election timeouts (force this in code temporarily). 2. Stop any initial leader to force election. 3. Observe logs for multiple election cycles.	Initially, all nodes may become `Candidate` simultaneously, causing split vote (no majority). After randomized retry, one eventually wins.
Term synchronization	Nodes discovering higher term should step down.	1. Start three nodes, let node 1 become leader term 1. 2. Manually set node 2's `currentTerm` to 10 (simulate restart with persisted state). 3. Node 2 receives `AppendEntries` from node 1 (term 1). 4. Observe node 2's response.	Node 2 rejects RPC due to higher term, includes `Term: 10` in reply. Node 1 sees higher term, steps down to `Follower` and updates its term to 10.

Verification Commands: Create a simple test harness that starts nodes and captures their logs. For a three-node test:

```bash
# Build and run three nodes in separate terminals or background processes

go run cmd/node/main.go --id 1 --peers 2,3 --port 8080

go run cmd/node/main.go --id 2 --peers 1,3 --port 8081

go run cmd/node/main.go --id 3 --peers 1,2 --port 8082
```

Expected log snippets (your actual logs will vary):

```
Node 1: [Term 1] Election timeout, transitioning to Candidate
Node 1: [Term 1] Sending RequestVote to Node 2
Node 2: [Term 1] Granted vote to Node 1
Node 1: [Term 1] Received majority votes, becoming Leader
Node 2: [Term 1] Received AppendEntries heartbeat from Leader 1
Node 3: [Term 1] Received AppendEntries heartbeat from Leader 1
```

Milestone 2: Log Replication Verification

This milestone verifies that the leader can replicate log entries to followers, handle consistency checks, and advance the commit index when entries are replicated to a majority.

Test Scenario	Expected Behavior	Verification Steps	Pass/Fail Criteria
Single entry replication	Leader replicates one entry to all followers, entry becomes committed.	1. Start three nodes, elect leader. 2. Leader receives client command "SET X=5". 3. Leader appends entry, sends `AppendEntries`. 4. Observe logs for 1 second.	All nodes have entry in log with same term/index. Leader's `commitIndex` advances to 1 after majority acknowledgment. Followers' `commitIndex` updates via next heartbeat.
Multiple consecutive entries	Sequence of entries maintains order and consistency.	1. With leader established, submit commands "SET A=1", "SET B=2", "SET C=3". 2. Observe replication flow.	All nodes' logs show entries [1,2,3] in order with correct terms. Commit index advances stepwise as each entry reaches majority.
Follower log inconsistency recovery	Leader brings follower's log into consistency using `prevLogIndex` / `prevLogTerm` and retries.	1. Start three nodes, replicate entry 1 (term 1) to all. 2. Isolate follower 3 (network partition). 3. Leader replicates entries 2-4 (term 1) to remaining follower only. 4. Heal partition, leader sends next `AppendEntries` to follower 3. 5. Observe consistency check and potential log truncation.	Follower 3 rejects first `AppendEntries` due to mismatched `prevLogTerm`. Leader decrements `nextIndex[3]` and retries with earlier entries, eventually overwriting inconsistent entries.
Heartbeat maintenance	Empty `AppendEntries` (heartbeats) prevent followers from starting elections.	1. Establish leader. 2. No client commands for 2+ election timeout periods. 3. Observe RPC traffic.	Leader sends periodic heartbeats. Followers reset election timers on each heartbeat, remain followers. No election occurs.
Commit index propagation	Leader's commit index is propagated to followers via subsequent `AppendEntries`.	1. Replicate entry 1 to majority but not all followers. 2. Leader advances `commitIndex` to 1. 3. Leader sends next heartbeat or new entry. 4. Check followers' `commitIndex`.	Followers update their `commitIndex` to min(`LeaderCommit`, `lastLogIndex`). All followers eventually apply entry 1 to state machine.

Verification Commands: Extend your test harness to submit commands to the leader:

```bash
# Submit a command via HTTP (if you've implemented a simple client interface)

curl -X POST http://localhost:8080/command -d '{"cmd":"SET X=5"}'
```

Expected log snippets:

```
Leader Node 1: [Term 1] Received client command: SET X=5
Leader Node 1: [Term 1] Appending entry at index 1, term 1
Leader Node 1: [Term 1] Sending AppendEntries to Node 2, prevLogIndex=0, prevLogTerm=0
Follower Node 2: [Term 1] AppendEntries success, added 1 entry
Leader Node 1: [Term 1] Entry replicated to majority, commitIndex=1
Leader Node 1: [Term 1] Applied entry 1 to state machine: SET X=5
Follower Node 2: [Term 1] LeaderCommit=1, updating commitIndex to 1
Follower Node 2: [Term 1] Applied entry 1 to state machine: SET X=5
```

Milestone 3: Safety Properties Verification

Safety properties are often violated in edge cases. These tests specifically target scenarios that could break Raft's guarantees if implemented incorrectly.

Test Scenario	Expected Behavior	Verification Steps	Pass/Fail Criteria
Election Safety (no two leaders same term)	Never two leaders in same term, even during network partitions.	1. Start five nodes. 2. Create network partition isolating 3 nodes from 2. 3. Allow election on both sides. 4. Heal partition after both sides may have leaders. 5. Observe term reconciliation.	Only the leader in the higher term survives. The lower-term leader steps down when it sees a higher term. At no point do two nodes believe they are leader for the same term.
Leader Completeness (committed entries not lost)	Any entry committed in a term must be present in future leaders' logs.	1. Start five nodes, elect leader L1 term 1. 2. Replicate entry 1 to majority (committed). 3. Crash L1 and a follower that has entry 1. 4. Remaining three nodes elect new leader L2 term 2. 5. Check L2's log.	L2's log contains entry 1 from term 1 (committed entry survives). The `RequestVote` RPC's log comparison ensures only nodes with up-to-date logs can become leader.
State Machine Safety (same apply order)	All nodes apply exactly the same entries in same order.	1. Start three nodes. 2. Submit concurrent commands from multiple clients. 3. Introduce random delays and packet loss. 4. After 100 commands, stop all nodes. 5. Compare each node's applied command sequence.	All nodes have identical sequences of applied commands (same commands, same order). Logs may have uncommitted divergent entries, but applied sequences match exactly.
Figure 8 scenario prevention	Leader does not commit entries from previous terms until its own entry commits.	1. Create scenario from Raft paper Figure 8 (see diagram ). 2. Simulate leader crashes and elections. 3. Check that entry from term 2 is not committed unless it is also replicated in term 3.	The safety rule (leader only commits entries from current term after they are replicated) prevents inconsistency. No node applies entry from term 2 unless it's also present in leader's term 3 log.
Log Matching Property	If two logs have same index and term, they are identical up to that index.	1. Create divergent logs on two followers via isolation and partial replication. 2. Heal network, allow leader to reconcile. 3. Compare logs after reconciliation.	After consistency check, logs become identical up to the last common index. The `AppendEntries` consistency check (`prevLogIndex` , `prevLogTerm`) ensures this property.

Verification Commands: For safety tests, you'll need more controlled simulation. Consider using a testing framework that lets you inject failures:

```bash
# Run a scenario test that simulates Figure 8

go test ./tests/ -run TestFigure8Scenario -v
```

Expected outcome: Test passes without errors. To verify Leader Completeness manually:

```
# After leader crash and new election, check new leader's log
New Leader Log:
Index 1: Term 1, Command SET X=5 (committed)
Index 2: Term 2, Command SET Y=10 (not committed, may be overwritten)
```

Milestone 4: Cluster Membership Changes Verification

Membership changes introduce complexity; tests must verify safety during configuration transitions.

Test Scenario	Expected Behavior	Verification Steps	Pass/Fail Criteria
Single server addition (C→C+1)	New server can be added safely, catch-up before voting.	1. Start 3-node cluster. 2. Propose configuration change to add server 4 as learner. 3. After server 4 catches up, propose change to make it voting member. 4. Observe joint consensus steps.	Configuration entries are replicated as log entries. During transition, both old and new configurations are considered for majority. New server does not vote until fully caught up.
Single server removal (C→C-1)	Removed server stops participating in consensus.	1. Start 4-node cluster. 2. Propose removal of server 4. 3. After configuration committed, send RPCs from leader to removed server. 4. Observe behavior.	Removed server rejects `AppendEntries` or is not contacted. Leader's `ConfigState` no longer includes removed server in `VotingMembers()`.
Joint consensus safety	No disjoint majorities can decide during transition.	1. Start configuration C_old (servers 1,2,3). 2. Start transition to C_new (servers 3,4,5). 3. During joint consensus (C_old,new), simulate network partition separating old and new majorities. 4. Attempt to elect leaders on both sides.	No leader can be elected because neither side has majority of both configurations. System stalls until partition heals, maintaining safety.
Configuration entry commitment	Configuration changes are committed like normal entries.	1. During membership change, track commit index. 2. After configuration entry is committed, verify it's applied to all nodes. 3. Crash and restart nodes.	Restarted nodes load committed configuration from log. Cluster maintains same configuration after recovery.
Availability during changes	Cluster remains available for client operations during changes.	1. While configuration change is in progress, submit client commands. 2. Measure latency and success rate. 3. Verify commands are replicated and committed correctly.	Client commands succeed (if they reach leader). Configuration change does not block normal operation indefinitely.

Verification Commands: Test membership changes with a script that proposes configurations:

```bash
# Add server 4 to a 3-node cluster

curl -X POST http://localhost:8080/config -d '{"type":"ADD", "server":4, "address":"localhost:8083"}'
```

Expected log snippets:

```
Leader Node 1: [Term 5] Proposing joint consensus configuration: Old=[1,2,3], New=[1,2,3,4]
Leader Node 1: [Term 5] Replicating config entry at index 24
Node 2: [Term 5] Config entry replicated, entering joint consensus state
Leader Node 1: [Term 5] Joint consensus entry committed
Leader Node 1: [Term 5] Proposing new configuration: [1,2,3,4]
Node 4: [Term 5] Caught up to log index 24, now voting member
```

11.2 Property-Based Testing Approach

Property-based testing generates random inputs and system conditions to verify that certain invariants always hold. This approach is particularly powerful for distributed systems because it can uncover edge cases that manual scenario testing might miss. The core idea is to define **invariants** (properties that must always be true) and **generators** (random sequences of events like network partitions, node crashes, message delays), then run thousands of simulations checking that invariants hold.

Safety Invariants to Test

These properties must hold at all times during execution. Implement them as assertion functions that can be called at any point during a simulation.

Invariant Name	Formal Description	How to Verify in Code
Election Safety	For any term T, at most one `RaftNode` has `state == Leader` and `currentTerm == T`.	Periodically collect all nodes' (term, state) pairs. Assert no two nodes have (Leader, same term).
Leader Completeness	If a log entry is committed in term T, then it will be present in the log of every leader for all terms > T.	When entry commits, record its (index, term). When new leader elected, verify its log contains all recorded committed entries.
State Machine Safety	If a node has applied a log entry at index I, no other node may apply a different entry at the same index.	Compare `lastApplied` entries across nodes. All nodes with `lastApplied >= I` must have identical entries for indices ≤ I.
Log Matching Property	If two logs contain an entry with the same index and term, then the logs are identical in all entries up to that index.	Compare logs pairwise. For any common index where terms match, assert all preceding entries match.
Term Monotonicity	A node's `currentTerm` never decreases.	Track term history per node, assert each new term > previous term.
Leader Append-Only	A leader never overwrites or deletes entries in its own log; only followers may truncate logs during consistency check.	Leader's log should only grow via `Append`. Check that `LogStore.LastIndex()` never decreases for a leader.
Commit Monotonicity	`commitIndex` never decreases on any node.	Track `commitIndex` over time, assert it only increases or stays same.
Applied ≤ Committed	`lastApplied` ≤ `commitIndex` at all times.	Simple assertion after each state update.
Vote Safety	A node grants at most one vote per term.	Check `votedFor` is not changed within same term after first vote.
Quorum Intersection	Any two majorities (quorums) in the same configuration must intersect.	For configuration C, compute all possible majority subsets; verify every pair has non-empty intersection.

Liveness Properties to Test

Liveness properties state that something good eventually happens (e.g., a leader is elected). These are harder to test conclusively but can be verified probabilistically.

Property	Description	Testing Approach
Leader Election Liveness	If a majority of servers are reachable and can communicate, eventually a leader will be elected.	In simulations with stable network, measure time to first leader election. Should happen within few election timeout cycles.
Log Replication Liveness	If a leader is stable and a majority of followers are reachable, a client command will eventually be committed.	Submit commands during stable periods, verify they commit within reasonable time.
Configuration Change Liveness	A configuration change proposed by a stable leader will eventually complete.	Propose changes during stable periods, track time to completion.

Building a Randomized Testing Framework

Create a simulation framework that virtualizes time and network to run deterministic tests. This is similar to the approach used in the Raft paper's formal verification and tools like Jepsen.

Architecture Decision: Simulation vs. Real Network

> **Decision: Use Deterministic Simulation for Property-Based Tests**
>
> - **Context**: We need to run thousands of test iterations with random faults but require reproducibility for debugging failures.
> - **Options Considered**:
>   1. **Real network with fault injection**: Use actual sockets/networking with random delays, drops, and partitions. More realistic but non-deterministic and slow.
>   2. **Deterministic virtual time simulation**: Model nodes, network, and time in a single process using virtual clocks and mocked communication. Deterministic and fast but requires implementing a simulation framework.
> - **Decision**: Build a deterministic simulation framework.
> - **Rationale**: Reproducibility is critical when a test fails—we need to replay the exact sequence of events to debug. Speed allows running thousands of iterations. The simulation can model realistic network behaviors (partitions, delays) while maintaining control.
> - **Consequences**: Need to implement virtual time, a simulated network layer, and node wrappers. Tests will run in a single process without real networking.

Option	Pros	Cons	Chosen?
Real network with fault injection	Tests real networking code, more confidence	Non-deterministic failures hard to debug, slow execution, complex setup	No
Deterministic simulation	Reproducible failures, fast execution, full control over events	Requires simulation framework, may not catch real networking bugs	**Yes**

Simulation Framework Components:

1. **Virtual Time**: Replace `time.Sleep` and timers with a virtual clock that can be advanced manually.
2. **Simulated Network**: Replace the `Transport` interface with an in-memory implementation that can delay, reorder, or drop messages based on rules.
3. **Node Wrappers**: Wrap `RaftNode` instances to intercept their timer calls and RPC sends.
4. **Event Scheduler**: Schedule events (node crashes, partition changes, client requests) at specific virtual times.
5. **Checker**: Periodically check invariants and record violations.

Example Test Structure:

```go
func TestRaftSafetyRandomized(t *testing.T) {

    rand.Seed(42) // Fixed seed for reproducibility



    for i := 0; i < 1000; i++ { // Run 1000 random simulations

        sim := NewSimulator(5) // 5-node cluster



        // Random events: partitions, crashes, restarts

        sim.SchedulePartition(10, []int{1,2}, []int{3,4,5}) // At time 10, split cluster

        sim.ScheduleCrash(15, 3) // At time 15, crash node 3

        sim.ScheduleRecover(25, 3) // At time 25, recover node 3

        sim.ScheduleHealPartition(30) // At time 30, heal partition



        // Submit random client commands

        for j := 0; j < 20; j++ {

            cmd := randomCommand()

            sim.ScheduleCommand(5+j*2, cmd) // Submit at increasing times

        }



        // Run simulation for virtual 60 seconds

        sim.Run(60)



        // Verify all invariants

        if err := sim.CheckAllInvariants(); err != nil {

            t.Errorf("Iteration %d failed: %v", i, err)

            sim.PrintTrace() // Output event trace for debugging

            break

        }

    }

}
```

Common Random Faults to Inject:

- **Network partitions**: Split cluster into two or more groups that cannot communicate.
- **Message loss**: Randomly drop RPCs (e.g., 10% loss rate).
- **Message delay**: Delay RPCs by random intervals (e.g., 0-100ms).
- **Node crashes**: Stop a node (freeze its state) for a period, then restart it with persisted state.
- **Unreliable timing**: Vary election timeouts and heartbeat intervals slightly.

Interpreting Test Results: When a property-based test fails:

1. **Reproduce**: Re-run with the same random seed.

2. **Trace**: Examine the detailed event trace leading to the failure.
3. **Minimize**: Try to reduce the sequence of events to a minimal failing case.
4. **Debug**: Use added diagnostic logging to understand node states at the failure point.

Implementation Guidance

Technology Recommendations:

Component	Simple Option	Advanced Option
Unit Testing	Go's built-in `testing` package	Testify framework for assertions and mocks
Property-Based Testing	Manual random simulation as described	Use `rapid` or `gopter` for property-based testing
Network Simulation	In-memory channel-based transport	Use `gnet` or custom virtual network layer
State Inspection	JSON output via HTTP endpoint	Integrated visualization similar to raft.github.io

Recommended File Structure for Tests:

```
project-root/
  internal/raft/           # Core Raft implementation
    raft.go                # RaftNode implementation
    raft_test.go           # Unit tests for individual methods
  tests/                   # Integration and property tests
    simulation/            # Simulation framework
      simulator.go         # Virtual time and network simulator
      virtual_transport.go # Simulated transport
      invariants.go        # Invariant checking functions
    scenarios/             # Predefined scenario tests
      leader_election_test.go
      log_replication_test.go
      safety_test.go
      membership_test.go
    property/              # Property-based tests
      randomized_test.go   # Main randomized test
      invariants_test.go   # Invariant verification
  cmd/                     # Executables
    node/                  # Main node binary
    test_runner/           # Special test runner with visualization
```

Infrastructure Starter Code: Virtual Transport for Testing:

```go
// tests/simulation/virtual_transport.go

package simulation

import (
    "sync"
    "time"
)

// VirtualTransport simulates network with delays, drops, and partitions.
type VirtualTransport struct {
    mu          sync.RWMutex
    nodes       map[int]*VirtualEndpoint
    dropRate    float64 // 0.0 to 1.0
    minDelay    time.Duration
    maxDelay    time.Duration
    partitions  [][]int // Groups that can communicate only internally
    clock       *VirtualClock
}

type VirtualEndpoint struct {
    id      int
    inbox   chan Message
    handler func(Message) (interface{}, error)
}

func NewVirtualTransport(clock *VirtualClock) *VirtualTransport {
    return &VirtualTransport{
        nodes:    make(map[int]*VirtualEndpoint),
        clock:    clock,
        minDelay: 0,
        maxDelay: 50 * time.Millisecond,
    }
}

func (vt *VirtualTransport) Register(nodeID int, handler func(Message) (interface{}, error)) {
    vt.mu.Lock()
    defer vt.mu.Unlock()
    vt.nodes[nodeID] = &VirtualEndpoint{
```

```
        id:      nodeID,
        inbox:   make(chan Message, 1000),
        handler: handler,
    }
}

func (vt *VirtualTransport) Send(from, to int, msg Message) (interface{}, error) {
    vt.mu.RLock()
    defer vt.mu.RUnlock()

    // Check if message should be dropped
    if vt.shouldDrop() {
        return nil, errors.New("simulated network drop")
    }

    // Check partition restrictions
    if !vt.canCommunicate(from, to) {
        return nil, errors.New("partitioned")
    }

    target, ok := vt.nodes[to]
    if !ok {
        return nil, errors.New("node not found")
    }

    // Calculate delay
    delay := vt.randomDelay()

    // Schedule delivery with virtual clock
    vt.clock.AfterFunc(delay, func() {
        // Deliver message by calling handler directly (simulating RPC)
        go func() {
            resp, err := target.handler(msg)
            if err == nil && msg.Type == "RequestVote" {
                // Send response back (simplified)
                if src, ok := vt.nodes[from]; ok {
```

```
                    src.handler(Message{
                        Type: "RequestVoteReply",
                        From: to,
                        To:   from,
                        Payload: encodeResponse(resp),
                    })
                }
            }
        }()
    })

    return nil, nil
}

func (vt *VirtualTransport) shouldDrop() bool {
    return vt.dropRate > 0 && rand.Float64() < vt.dropRate
}

func (vt *VirtualTransport) canCommunicate(a, b int) bool {
    if len(vt.partitions) == 0 {
        return true
    }
    for _, group := range vt.partitions {
        aInGroup := false
        bInGroup := false
        for _, id := range group {
            if id == a { aInGroup = true }
            if id == b { bInGroup = true }
        }
        if aInGroup && bInGroup {
            return true
        }
    }
    return false
}
```

Core Logic Skeleton for Invariant Checking:

```go
// tests/simulation/invariants.go

package simulation

import (
    "fmt"
    "strings"
)

// CheckElectionSafety verifies at most one leader per term.
func CheckElectionSafety(nodes map[int]*RaftNode) error {
    leadersByTerm := make(map[int][]int)

    for id, node := range nodes {
        node.mu.Lock()
        if node.state == Leader {
            leadersByTerm[node.currentTerm] = append(leadersByTerm[node.currentTerm], id)
        }
        node.mu.Unlock()
    }

    for term, leaders := range leadersByTerm {
        if len(leaders) > 1 {
            return fmt.Errorf("election safety violated in term %d: multiple leaders %v", term, leaders)
        }
    }
    return nil
}

// CheckStateMachineSafety verifies all nodes have same applied entries.
func CheckStateMachineSafety(nodes map[int]*RaftNode) error {
    // Collect applied entries from each node
    applied := make(map[int][]LogEntry)
    maxIndex := 0

    for id, node := range nodes {
        node.mu.Lock()
        // TODO 1: Get all applied entries from node's state machine or log
```

```
        // Hint: You might need to expose node.lastApplied and node.log
        // applied[id] = node.log[:node.lastApplied]
        // maxIndex = max(maxIndex, node.lastApplied)
        node.mu.Unlock()
    }

    // TODO 2: Compare applied entries across all nodes up to maxIndex
    // For index from 1 to maxIndex:
    //   For each pair of nodes (id1, id2):
    //     if applied[id1][index] != applied[id2][index] {
    //         return error
    //     }

    return nil
}

// CheckLeaderCompleteness verifies committed entries survive leader changes.
func CheckLeaderCompleteness(nodes map[int]*RaftNode, committedEntries map[int]LogEntry) error {
    // TODO 1: When an entry commits during simulation, record it in committedEntries
    // TODO 2: When a new leader is elected, check its log contains all committedEntries
    // for index, expectedEntry := range committedEntries {
    //     if leader.log[index] != expectedEntry {
    //         return error
    //     }
    // }
    return nil
}
```

Milestone Checkpoint Test Commands:

For each milestone, create a dedicated test file:

```bash
# Run Milestone 1 tests
go test ./tests/scenarios/ -run TestLeaderElection -v

# Run Milestone 2 tests
go test ./tests/scenarios/ -run TestLogReplication -v

# Run Milestone 3 safety tests
go test ./tests/scenarios/ -run TestSafetyProperties -v

# Run Milestone 4 membership tests
go test ./tests/scenarios/ -run TestMembershipChanges -v

# Run all randomized property tests (might take minutes)
go test ./tests/property/ -v
```

Debugging Tips for Test Failures:

Symptom	Likely Cause	How to Diagnose	Fix
Elections never complete	Election timeout too short/long, votes not being granted	Check `RequestVote` RPC logs: are terms correct? Are logs "up-to-date"?	Ensure `LastLogTerm` and `LastLogIndex` comparisons are correct. Check random timeout generation.
Split votes keep happening	Randomization range too small, nodes restart elections simultaneously	Log election timeout values for each node. Are they truly random?	Increase randomization range (e.g., 150-300ms instead of 150-200ms).
Log entries disappear after leader change	Leader completeness violation, `RequestVote` log comparison wrong	Check new leader's log vs old leader's committed entries.	Ensure `RequestVote` grants vote only if candidate's log is at least as up-to-date.
Commit index doesn't advance	Majority calculation wrong, `matchIndex` not updated properly	Log `matchIndex` array on leader after each `AppendEntries` response.	Verify majority calculation includes leader itself. Check `nextIndex` / `matchIndex` update logic.
Nodes apply different commands at same index	Figure 8 scenario, committing entries from old terms	Check if leader commits entry before replicating its own term entry.	Implement safety rule: leader only commits entries from current term after they're replicated.
Configuration change gets stuck	Joint consensus not implemented correctly, disjoint majorities	Log configuration state on each node. Check if any majority exists.	Ensure joint consensus requires majority from BOTH old and new configurations.
Property-based test fails intermittently	Race condition, timing issue, network partition edge case	Re-run with same seed, add detailed logging at failure point.	Examine minimal failing case. May need to adjust timeouts or add synchronization.

Language-Specific Hints (Go):

- Use `t.Parallel()` carefully in Raft tests—nodes share simulated network but real concurrency can cause non-determinism.
- For deterministic randomness, create a seeded `rand.Rand` instance per simulation.
- Use `sync.Map` for simulation state that's accessed by multiple goroutines.
- Add `-race` flag to catch data races: `go test -race ./...`
- Use `go test -timeout 30s` for longer-running property tests.

12. Debugging Guide

> **Milestone(s):** 1 (Leader Election), 2 (Log Replication), 3 (Safety Properties), 4 (Cluster Membership Changes)

Debugging a distributed consensus algorithm is significantly more challenging than debugging a single-threaded application. The system's behavior emerges from the interaction of multiple independent nodes, each with its own concurrent internal state, communicating over an unreliable network. Failures are not just possible but expected, and bugs often manifest as subtle violations of safety properties that only appear under specific timing conditions or failure sequences. This section provides a structured approach to diagnosing and resolving common issues in a Raft implementation.

12.1 Common Bug Symptoms and Fixes

The following table catalogs the most frequent symptoms encountered during Raft implementation, their root causes, and step-by-step fixes. Each symptom is a visible manifestation of an underlying protocol violation.

Symptom	Likely Cause(s)	Diagnostic Steps	Fix
Elections never complete – Nodes oscillate between `Follower` and `Candidate` indefinitely, no stable `Leader` emerges.	1. **Split votes**: Multiple candidates with equally up-to-date logs trigger simultaneous elections, none achieves majority. 2. **Insufficient peers for quorum**: The cluster size is too small for the failure scenario (e.g., 2 of 3 nodes are down). 3. **Election timer reset bugs**: The election timeout is reset incorrectly (e.g., on any RPC, not just valid ones), preventing timeouts from ever firing. 4. **Vote granting logic too strict**: The `RequestVote` handler rejects candidates due to an overly restrictive log comparison.	1. Check logs for multiple candidates incrementing term and requesting votes in the same term. 2. Verify cluster size and which nodes are reachable. Check if a partition isolates candidates from a majority. 3. Add logging for every `resetElectionTimeout()` call. Ensure it's only called on receiving a valid `AppendEntries` from a current leader or granting a vote. 4. Inspect the `LastLogTerm` and `LastLogIndex` values in `RequestVoteArgs` and compare with the follower's log. Verify the `logIsAtLeastAsUpToDate` logic.	1. Ensure election timeout randomization is implemented correctly (e.g., 150-300ms range). This reduces simultaneous candidacy probability. 2. For a 3-node cluster, at least 2 must be operational. Ensure network connectivity. 3. Only reset the election timer when: a) receiving an `AppendEntries` RPC from a current leader (term >= currentTerm), or b) granting a vote to a candidate (term > currentTerm). 4. Re-read Raft paper Section 5.4.1: a log is more up-to-date if its last term is higher, or if terms are equal, its index is higher or equal. Fix the comparison.
Leader cannot replicate entries – The leader sends `AppendEntries` but followers consistently reply with `Success = false`.	1. **Incorrect `PrevLogIndex` / `PrevLogTerm`**: The leader's `nextIndex` for a follower is wrong, pointing to a non-existent index or a term mismatch. 2. **Log inconsistency after leader crash**: A new leader has a different log history than followers, and the consistency check fails. 3. **Term confusion in `AppendEntries` handler**: The follower rejects entries because it sees a term mismatch but doesn't update its own term properly.	1. Log the `PrevLogIndex` and `PrevLogTerm` sent by the leader and the follower's log at that index (if it exists). 2. Compare the logs of all nodes. A visual log diff tool is invaluable. 3. Check if the follower's `checkTermAndStepDown(term)` logic is implemented correctly. Does it update its `currentTerm` and revert to `Follower` when `args.Term > currentTerm`?	1. Implement **log reconciliation**: When `AppendEntries` fails, the leader should decrement `nextIndex[peer]` and retry with earlier log entries. Eventually, it will find a matching point. 2. Ensure the leader initializes `nextIndex` to its last log index + 1 after election (Raft paper Section 5.3). 3. In `handleAppendEntries`, if `args.Term > currentTerm`, update `currentTerm`, set `votedFor = nil`, and step down before processing the RPC.
Entries are committed but never applied – The `commitIndex` advances on the leader and followers, but `lastApplied` stays behind; the state machine doesn't see commands.	1. **Missing application loop**: No goroutine or periodic task calls `applyCommittedEntries()`. 2. **Race condition between commit and apply**: The `commitIndex` is updated but the application goroutine hasn't been notified. 3. **`lastApplied` not persisted**: After a crash, `lastApplied` resets to 0, but the log entries before the crash were already applied. Re-applying them causes duplicate side effects.	1. Verify that a dedicated goroutine or a timer-triggered function periodically calls `applyCommittedEntries()`. 2. Check if updates to `commitIndex` signal a condition variable or channel to wake the applier. 3. Inspect logs: after restart, does the node re-apply old entries?	1. Implement an **application goroutine** that loops, waiting for `commitIndex > lastApplied` (via condition variable or channel notification from `advanceCommitIndex`). 2. Use a **thread-safe mechanism** (e.g., `sync.Cond` or buffered channel) to notify the applier when `commitIndex` changes. 3. **`lastApplied` is volatile state**; it's okay to re-apply after a restart. The state machine must be **idempotent** or the application must track the last applied index externally.
Two leaders in the same term – Multiple nodes believe they are `Leader` for the same term, violating **Election Safety**.	1. **Missing step-down on higher term**: A leader does not revert to `Follower` upon receiving an RPC with a higher term. 2. **Network partition with old leader**: A partitioned former leader continues operating, unaware of a	1. Check logs: does a leader receive an `AppendEntries` or `RequestVote` with higher term and ignore it? 2. Simulate a network partition. Does the isolated leader eventually step down due to election timeout?	1. Implement `checkTermAndStepDown(term)` called at the start of every RPC handler. If `term > currentTerm`, set `currentTerm = term`, set `state = Follower`, and reset `votedFor = nil`.

Symptom	Likely Cause(s)	Diagnostic Steps	Fix
	new election in the majority partition. 3. **Race condition in election**: Two nodes collect votes for the same term due to stale `RequestVote` replies.	3. Examine `RequestVote` handling: does a node grant a vote without properly checking if it has already voted in this term?	2. This is expected behavior during a partition. The isolated leader will continue until it tries to communicate with the majority and discovers a higher term. Clients talking to it may get stale data. 3. Ensure the `votedFor` check in `handleRequestVote` is atomic with the term check. If `currentTerm == args.Term` and `votedFor` is already set (to another candidate), reject the vote.
Log entries disappear after leader change – Entries that were present in a previous leader's log are missing after a new leader is elected, violating **Leader Completeness**.	1. **Figure 8 scenario**: A leader committed an entry from an old term, then crashed. A new leader with a conflicting log overwrote it before it was replicated. 2. **Incorrect commit index advancement**: The leader advanced `commitIndex` for an entry that wasn't stored on a majority, then crashed. 3. **`nextIndex` backtracking too far**: During log reconciliation, the new leader decrements `nextIndex` past the point where the follower's log matches, causing valid entries to be overwritten.	1. Reproduce the Figure 8 scenario from the Raft paper. Check if the safety rule (Section 5.4.2) is implemented: a leader only counts replicas for entries from its own term when advancing `commitIndex`. 2. Audit `advanceCommitIndex`: does it require a majority of `matchIndex[i] >= N` for each index N? 3. When `AppendEntries` fails, the leader should decrement `nextIndex` by 1, but not below 1 (log index starts at 1).	1. Implement the **safety rule**: In `advanceCommitIndex`, only consider entries from the leader's current term for commitment. Once an entry from the current term is committed, all prior entries are committed indirectly. 2. Ensure `matchIndex` is updated correctly when an `AppendEntries` succeeds. The leader sets `matchIndex[peer] = args.PrevLogIndex + len(args.Entries)`. 3. Bound `nextIndex` decrement to a minimum of 1. Alternatively, implement the more efficient optimization in the Raft paper (Section 5.3) using conflict term and index.
Cluster gets stuck during membership change – After proposing a configuration change, the cluster cannot commit new entries or elect a leader.	1. **Disjoint majorities**: The old and new configurations do not overlap sufficiently, allowing two separate majorities to form (one in old, one in new). 2. **Leader not in new configuration**: The leader that proposed the change is removed in the new configuration and steps down before the change commits. 3. **Missing joint consensus handling**: The implementation attempts single-step changes, which can cause safety violations.	1. Check the configuration log entries. Are both the old and new configurations correctly logged? 2. Verify that the leader remains in both configurations during joint consensus (`C_old,new`). 3. Does the `ConfigState.QuorumMajority` function require majorities from both configurations when in joint consensus?	1. Always use **joint consensus** for membership changes. The intermediate configuration `C_old,new` requires a majority from both `C_old` and `C_new` for commitment. 2. Ensure the leader includes itself in the new configuration if it is to remain a voting member. Leaders removed should step down only after `C_new` is committed. 3. Implement `ConfigState.IsJoint()`, `ConfigState.VotingMembers()`, and `ConfigState.QuorumMajority()` to correctly compute quorums for joint consensus.
High CPU usage in leader replication loop – The leader's `broadcastAppendEntries` or `startReplicationLoop`	1. **Busy-wait loop**: The replication loop lacks a delay or condition to throttle sending RPCs. 2. **Aggressive retries**: Failed RPCs are retried immediately without backoff, causing tight loops.	1. Profile the CPU usage to confirm the replication loop is the source. 2. Check if the loop has a `time.Sleep` or is triggered by a timer/heartbeat interval.	1. Implement a **heartbeat interval** (e.g., 50ms). The leader should send `AppendEntries` (heartbeats or with entries) at this interval, not in a tight loop. 2. For retries, use exponential backoff

Symptom	Likely Cause(s)	Diagnostic Steps	Fix
consumes excessive CPU by spinning without waiting.	3. **Excessive logging**: Verbose logging in the hot path slows down the system and creates log spam.	3. Examine log output for rapid retry messages.	or wait for the next heartbeat interval. 3. Use structured logging with log levels (e.g., debug, info). Disable debug logs in production simulations.
Node crashes and restarts with empty log – After a crash and restart, the node's persistent log (`[]LogEntry`) is empty, losing all committed state.	1. **Missing persistence writes**: The `PersistentState` (currentTerm, votedFor, log) is not written to stable storage before replying to RPCs. 2. **Crash at inopportune time**: The node crashed after appending to memory but before persisting. 3. **Incorrect recovery**: The `recoverNode()` function fails to read the persisted state.	1. Add logging after every persistence call. Verify that `PersistentState` is saved after each change to `currentTerm` , `votedFor` , or `log` . 2. Check the order of operations: log append -> persist -> send RPC reply. 3. Verify the persistence layer (e.g., file I/O) works correctly and the data format is consistent.	1. Implement **persistence hooks** that save to stable storage after any change to `currentTerm` , `votedFor` , or `log` . Call these hooks in `RaftNode` methods before releasing locks. 2. Follow the Raft paper's rule: "Before responding to an RPC, a server must persist information that is required for safety." 3. Ensure `recoverNode()` reads the same file/format and initializes the `RaftNode` fields. Add checksum or version to detect corruption.

12.2 Debugging Techniques for Raft

Effective debugging of a distributed system requires moving beyond printf statements. The following techniques will help you visualize state, trace interactions, and systematically reproduce failures.

Structured Logging with Context

> **Mental Model: The Air Traffic Control Transcript** – Imagine each Raft node as an air traffic controller tower. Every radio transmission (RPC), internal decision (state change), and observed event (timeout) is recorded in a timestamped transcript with a consistent format. When investigating an incident, you can merge transcripts from all towers to reconstruct the complete sequence of events.

Add a structured logging function to your `RaftNode` that captures essential context. This log should be designed for machine readability (e.g., JSON lines) to enable filtering and analysis.

Key fields to include in every log entry:

- `timestamp` : High-resolution time (nanoseconds)
- `node` : The node ID
- `term` : The node's current term at time of logging
- `state` : The node's state (`Follower` , `Candidate` , `Leader`)
- `event` : A descriptive event name (e.g., `election_timeout` , `rv_request` , `ae_reply`)
- `details` : A structured object with event-specific data (e.g., `{"candidateId": 3, "granted": true}`)

Where to add logging:

1. **State transitions:** When `RaftNode.state` changes.
2. **RPC entry/exit:** At the beginning and end of `handleRequestVote` and `handleAppendEntries` , including arguments and reply.
3. **Timer events:** When an election timeout fires, and when it's reset.
4. **Commit and apply:** When `commitIndex` or `lastApplied` changes, and when an entry is applied to the state machine.
5. **Persistent changes:** When `currentTerm` , `votedFor` , or `log` are saved.

Implementation pattern:

```go
func (rn *RaftNode) logEvent(event string, details map[string]interface{}) {
    // Use a logging library or write JSON to stderr
    log.Printf("RAFT %d T%d S%s: %s %v", rn.id, rn.currentTerm, rn.state, event, details)
}
```

State Inspector Tool

Build a simple HTTP or TCP endpoint on each node that returns a snapshot of its internal state in a structured format (JSON). This allows you to query node status without digging through logs. The endpoint should be accessible even if the node's main RPC transport is blocked or deadlocked.

Expose the following information:

- Node ID, current term, state, votedFor
- Log summary: last index, last term, commit index, last applied
- For leaders: `nextIndex` and `matchIndex` for each peer
- Current configuration (and pending if in joint consensus)
- Various timers: time since last heartbeat, election timeout remaining

You can implement this by adding an HTTP handler to your `HTTPTransport` or creating a separate debug server. This tool is invaluable for interactive debugging and writing automated health checks.

Deterministic Simulation and Trace Visualization

The most powerful technique for debugging consensus algorithms is **deterministic simulation**. By controlling randomness (e.g., seeding the random number generator), network conditions (message loss, delays, partitions), and node crashes, you can reproduce failures reliably. The `VirtualTransport` (introduced in the Testing Strategy) is designed for this.

Procedure:

1. **Record a failing test run:** When a property-based test fails, capture the seed value and the sequence of events (network decisions, crash points) that led to the failure.
2. **Replay the trace:** Re-run the test with the same seed, but add verbose logging at each step. Since the simulation is deterministic, the exact same failure will occur.
3. **Analyze the trace:** Use the structured logs from all nodes to create a **timeline visualization**. Tools like custom scripts or even manual diagramming can show the flow of RPCs, state changes, and log growth across nodes over simulated time.
4. **Identify the first deviation from spec:** Compare the trace against the Raft paper's rules. Look for the first event where your implementation's behavior diverges from the protocol (e.g., a leader didn't step down on higher term, a vote was granted incorrectly).

> **Key Insight:** In distributed systems, the root cause of a failure is often temporally distant from the symptom. A missing persistence write in term 2 might cause a leadership conflict in term 5. Deterministic replay allows you to walk back in time from the symptom to the root cause.

Invariant Checking in Tests

Embed invariant checks directly in your test harness. After every simulated event (message delivery, timeout, crash), pause the simulation and verify that safety properties hold across all nodes. This turns a failing test from "something went wrong eventually" into "invariant X was violated at time T."

Implement invariant verification functions (as partially defined in naming conventions) and call them periodically:

- `CheckElectionSafety(nodes map[int]*RaftNode) error`: Scans all nodes; returns error if more than one node is `Leader` with the same term.
- `CheckStateMachineSafety(nodes map[int]*RaftNode) error`: Compares the sequence of commands applied by each node's state machine; they must be identical up to the minimum `lastApplied`.
- `CheckLeaderCompleteness(nodes map[int]*RaftNode, committedEntries map[int]LogEntry) error`: For any entry known to be committed (replicated to a majority in some term), verify it exists in the log of every current leader and future leader.

When an invariant fails, the test can immediately halt and dump the full state of all nodes, giving you a snapshot of the exact moment correctness was broken.

Minimizing Heisenbugs

Concurrency and timing bugs (Heisenbugs) are common in Raft. They disappear when you add logging because logging introduces delays that change thread scheduling.

Strategies to combat Heisenbugs:

1. **Use a deterministic scheduler in tests:** The `VirtualTransport` can serialize all actions, removing concurrency. This eliminates race conditions as a variable.
2. **Run tests with the race detector:** In Go, use `go test -race`. This will identify data races (concurrent unsynchronized access to memory), which are often the source of Heisenbugs.
3. **Stress testing:** Run your implementation under heavy load (many commands, rapid leader changes) for extended periods. If a bug is timing-dependent, increasing the number of events increases the probability of triggering it.
4. **Model checking:** For advanced debugging, consider using a formal model checker like TLA+ to specify your protocol and verify safety properties. This can uncover subtle edge cases that randomized testing might miss.

By combining structured logging, state inspection, deterministic simulation, and runtime invariant checks, you build a comprehensive debugging toolkit that transforms the opaque, emergent behavior of a distributed system into a traceable, analyzable sequence of events.

Implementation Guidance

A. Technology Recommendations Table

Component	Simple Option	Advanced Option
Structured Logging	`log.Printf` with JSON marshaling, output to stderr	Structured logging library (e.g., `slog` in Go 1.21+, `zerolog`, `logrus`) with file rotation and log levels
State Inspection	HTTP endpoint on debug port (e.g., `:8081/debug/raft`) returning JSON	Integrated admin API with metrics (Prometheus), health checks, and log stream access
Trace Visualization	Custom script to parse logs and generate timeline (ASCII or HTML)	Integration with distributed tracing systems (OpenTelemetry) with UI (Jaeger)
Deterministic Testing	Custom `VirtualTransport` with seeded random and event logging	Use existing simulation frameworks (e.g., `github.com/etcd-io/raft`'s `rafttest` package, Jepsen-like harness)

B. Recommended File/Module Structure

Add debugging utilities to a dedicated package or integrate into existing ones:

```
project-root/
  internal/raft/
    node.go                 # RaftNode with integrated logging
    transport.go            # HTTPTransport with debug endpoint
    logstore.go             # LogStore
  internal/debug/
    inspector.go            # HTTP handler for node state inspection
    logger.go               # Structured logging utilities
  internal/simulation/
    virtual_transport.go    # Deterministic network simulator
    simulator.go            # Test harness with invariant checks
  tests/
    property_test.go        # Property-based tests with failure injection
    scenario_test.go        # Specific failure scenario tests
```

C. Infrastructure Starter Code: Structured Logger

Here's a complete, ready-to-use structured logger for debugging:

GO

```go
// internal/debug/logger.go

package debug

import (
    "encoding/json"
    "fmt"
    "os"
    "sync"
    "time"
)

type LogLevel int

const (
    LevelDebug LogLevel = iota
    LevelInfo
    LevelWarn
    LevelError
)

type LogEvent struct {
    Timestamp time.Time              `json:"timestamp"`
    NodeID    int                    `json:"node"`
    Term      int                    `json:"term,omitempty"`
    State     string                 `json:"state,omitempty"`
    Level     string                 `json:"level"`
    Event     string                 `json:"event"`
    Details   map[string]interface{} `json:"details,omitempty"`
}

type Logger struct {
    mu     sync.Mutex
    nodeID int
    level  LogLevel
    output *os.File
}

func NewLogger(nodeID int, level LogLevel, output *os.File) *Logger {
    if output == nil {
```

```
        output = os.Stderr
    }
    return &Logger{nodeID: nodeID, level: level, output: output}
}

func (l *Logger) Log(term int, state string, level LogLevel, event string, details map[string]interface{}) {
    if level < l.level {
        return
    }
    levelStr := "DEBUG"
    switch level {
    case LevelInfo:
        levelStr = "INFO"
    case LevelWarn:
        levelStr = "WARN"
    case LevelError:
        levelStr = "ERROR"
    }
    event := LogEvent{
        Timestamp: time.Now(),
        NodeID:    l.nodeID,
        Term:      term,
        State:     state,
        Level:     levelStr,
        Event:     event,
        Details:   details,
    }
    l.mu.Lock()
    defer l.mu.Unlock()
    data, _ := json.Marshal(event)
    fmt.Fprintln(l.output, string(data))
}

// Convenience methods
func (l *Logger) Debug(term int, state, event string, details map[string]interface{}) {
    l.Log(term, state, LevelDebug, event, details)
}
```

```go
func (l *Logger) Info(term int, state, event string, details map[string]interface{}) {
    l.Log(term, state, LevelInfo, event, details)
}

func (l *Logger) Warn(term int, state, event string, details map[string]interface{}) {
    l.Log(term, state, LevelWarn, event, details)
}

func (l *Logger) Error(term int, state, event string, details map[string]interface{}) {
    l.Log(term, state, LevelError, event, details)
}
```

Integrate into `RaftNode`:

GO

```go
// internal/raft/node.go

type RaftNode struct {

    // ... existing fields ...

    logger *debug.Logger

}


func NewRaftNode(id int, peers []int, sm StateMachine, transport Transport) *RaftNode {

    // ...

    rn := &RaftNode{

        id:       id,

        peers:    peers,

        state:    Follower,

        logger:   debug.NewLogger(id, debug.LevelInfo, nil),

        // ...

    }

    // ...

    return rn

}


// Example usage in handleRequestVote

func (rn *RaftNode) handleRequestVote(args RequestVoteArgs) RequestVoteReply {

    rn.logger.Debug(rn.currentTerm, rn.state.String(), "rv_request_received",

        map[string]interface{}{

            "candidate": args.CandidateID,

            "candidateTerm": args.Term,

            "lastLogIndex": args.LastLogIndex,

            "lastLogTerm": args.LastLogTerm,

        })

    // ... logic ...

    reply := RequestVoteReply{Term: rn.currentTerm, VoteGranted: granted}

    rn.logger.Info(rn.currentTerm, rn.state.String(), "rv_request_replied",

        map[string]interface{}{

            "candidate": args.CandidateID,

            "granted": reply.VoteGranted,

            "reason": reason,

        })

    return reply
```

```
}
```

D. Core Logic Skeleton: State Inspector Endpoint

Add a debug HTTP handler to your transport layer:

```go
// internal/debug/inspector.go

package debug

import (
    "encoding/json"
    "net/http"
    "sync"
)

type NodeState struct {
    ID           int                    `json:"id"`
    Term         int                    `json:"term"`
    State        string                 `json:"state"`
    VotedFor     *int                   `json:"votedFor,omitempty"`
    CommitIndex  int                    `json:"commitIndex"`
    LastApplied  int                    `json:"lastApplied"`
    LogSummary   map[string]interface{} `json:"logSummary"`
    NextIndex    map[int]int            `json:"nextIndex,omitempty"`
    MatchIndex   map[int]int            `json:"matchIndex,omitempty"`
    ConfigState  interface{}            `json:"configState,omitempty"`
}

type Inspector struct {
    mu    sync.RWMutex
    nodes map[int]func() NodeState
}

func NewInspector() *Inspector {
    return &Inspector{nodes: make(map[int]func() NodeState)}
}

func (i *Inspector) RegisterNode(id int, stateGetter func() NodeState) {
    i.mu.Lock()
    defer i.mu.Unlock()
    i.nodes[id] = stateGetter
}

func (i *Inspector) ServeHTTP(w http.ResponseWriter, r *http.Request) {
    i.mu.RLock()
```

```
    defer i.mu.RUnlock()

    states := make(map[int]NodeState)

    for id, getter := range i.nodes {

        states[id] = getter()

    }

    w.Header().Set("Content-Type", "application/json")

    json.NewEncoder(w).Encode(states)

}


// In RaftNode, implement a state getter:

func (rn *RaftNode) GetStateForInspector() debug.NodeState {

    rn.mu.Lock()

    defer rn.mu.Unlock()

    state := debug.NodeState{

        ID:          rn.id,

        Term:        rn.currentTerm,

        State:       rn.state.String(),

        CommitIndex: rn.commitIndex,

        LastApplied: rn.lastApplied,

        LogSummary: map[string]interface{}{

            "lastIndex": rn.log.LastIndex(),

            "lastTerm":  rn.log.LastTerm(),

            "size":      rn.log.Size(),

        },

    }

    if rn.votedFor != nil {

        state.VotedFor = rn.votedFor

    }

    if rn.state == Leader {

        state.NextIndex = make(map[int]int)

        state.MatchIndex = make(map[int]int)

        for i, idx := range rn.nextIndex {

            state.NextIndex[i] = idx

        }

        for i, idx := range rn.matchIndex {

            state.MatchIndex[i] = idx
```

```
        }
    }
    // Add config state if implemented
    return state
}
```

E. Language-Specific Hints (Go)

- Use `go test -race` to detect data races. Fix all races—they are bugs, not false positives.
- For deterministic testing, seed `math/rand` with a known value: `rand.Seed(seed)`.
- Use `sync.Cond` for efficient waiting on `commitIndex` changes in the applier goroutine.
- The `context` package can help manage RPC timeouts and cancellations.
- Profile CPU usage with `pprof`: `import _ "net/http/pprof"` and start a debug server.

F. Debugging Tips Table (Extended)

For quick reference during implementation:

Symptom	Quick Diagnostic	Likely Fix
Nodes stuck in candidate	Check log for "rv_request_received" and "rv_request_replied". Are votes being granted?	Verify log comparison and `votedFor` persistence.
Leader can't replicate	Inspect leader's `nextIndex` vs follower's log via debug endpoint.	Implement log reconciliation (decrement `nextIndex` on failure).
Commit index doesn't advance	Check `matchIndex` values on leader. Is a majority reached for any index?	Ensure `matchIndex` is updated on successful `AppendEntries`.
Duplicate client requests applied	Check `lastApplied` after restart. Are entries being re-applied?	Make state machine idempotent or track applied index in stable storage.
High CPU in leader loop	Check log for repeated "broadcast_heartbeat" events with no delay.	Add heartbeat interval (e.g., `time.Sleep(heartbeatInterval)`).
Test fails randomly	Run with `-race` and capture seed. Re-run with same seed and increased logging.	Look for race conditions or timing-dependent logic.

G. Milestone Checkpoint for Debugging

After implementing the structured logger and debug inspector, verify they work:

1. **Start a 3-node cluster** using your implementation.
2. **Trigger a leader election** by not sending heartbeats.
3. **Check logs:** Each node should log state transitions, vote requests, and grant decisions.
4. **Query debug endpoint:** `curl http://node1:8081/debug/raft` should return JSON with all nodes' states.
5. **Inject a failure:** Temporarily partition a follower from the leader.
6. **Observe behavior:** The leader should repeatedly try to replicate. Logs should show `AppendEntries` failures and `nextIndex` decrementing.
7. **Heal partition:** The follower should catch up. Verify via debug endpoint that logs match.

This debugging infrastructure will save countless hours when you progress to more complex failure scenarios in later milestones.

13. Future Extensions

> **Milestone(s):** (builds upon all milestones)

Congratulations on implementing the core Raft consensus algorithm! You now have a functioning distributed system that can safely replicate state across multiple nodes. However, the journey to a production-ready system has only begun. This section explores advanced features and extensions that transform your educational implementation into a practical system capable of handling real-world workloads. Each extension addresses specific limitations or performance characteristics that become apparent when deploying Raft in production environments.

The extensions are presented in order of increasing complexity, with each building upon concepts from previous milestones. Implementing these will deepen your understanding of distributed systems engineering and prepare you for real-world deployment scenarios.

13.1 Possible Extension Projects

13.1.1 Log Compaction and Snapshotting

Mental Model: The Library Archivist

Imagine a library that keeps every edition of every newspaper ever published. Over decades, the shelves become impossibly full, yet patrons mostly need recent issues. An archivist solves this by periodically creating condensed summaries (snapshots) of the complete collection up to a certain date, then discarding the physical newspapers older than that snapshot. Patrons can read recent newspapers directly, while for historical context they consult the snapshot and then any newspapers published after it. This is exactly what log compaction achieves: it prevents the log from growing unbounded while preserving the complete system state.

Concept and Rationale

The Raft log grows indefinitely as commands are applied to the state machine. In long-running systems, this creates three problems:

1. **Storage exhaustion**: Disk space is finite
2. **Restart recovery time**: Rebooting a node requires replaying the entire log
3. **New node catch-up overhead**: Adding a fresh node requires transmitting the entire log history

Log compaction addresses these by periodically persisting the complete state machine state (a **snapshot**) at a particular log index, then discarding all log entries up to that index. The snapshot contains the entire application state as of that index, and the **last included index** and **last included term** preserve the Raft consistency information needed for log matching property verification.

Key Design Decisions

> **Decision: Snapshot Installation Protocol**
>
> - **Context**: When a leader needs to send log entries to a follower that has discarded them (because it has a newer snapshot), the leader must transfer the snapshot instead.
> - **Options Considered**:
>   1. **Separate InstallSnapshot RPC**: Dedicated RPC for snapshot transfer with chunking support
>   2. **Extended AppendEntries**: Modify AppendEntries to include snapshot data for small snapshots
>   3. **Out-of-band transfer**: Use separate file transfer mechanism (HTTP, SCP) then coordinate via RPC
> - **Decision**: Implement a separate `InstallSnapshot` RPC as specified in the Raft paper
> - **Rationale**: The Raft paper provides a well-specified protocol with chunking support for large snapshots. This ensures compatibility with other Raft implementations and handles arbitrarily large state machines.
> - **Consequences**: Requires adding a new RPC type, snapshot storage management, and careful coordination between log truncation and snapshot installation.

Option	Pros	Cons	Chosen?
Separate InstallSnapshot RPC	Standardized, handles large snapshots via chunking, clear separation of concerns	Additional RPC type to implement, more protocol complexity	**Yes**
Extended AppendEntries	Reuses existing RPC infrastructure, simpler for small state machines	Limited by RPC size constraints, mixes snapshot and log replication concerns	No
Out-of-band transfer	Can use efficient file transfer protocols, offloads from Raft transport	Complex coordination, harder to guarantee consistency during transfer	No

Implementation Considerations

1. **Snapshot creation trigger**: Typically based on log size (e.g., snapshot when log exceeds 1GB) or number of entries
2. **Snapshot isolation**: Must capture a consistent state machine view at a specific log index
3. **Concurrent access**: Snapshots should be created asynchronously without blocking normal operations
4. **Storage format**: Include metadata (last included index/term) and serialized state machine state
5. **Follower snapshot installation**: Leaders send snapshots to lagging followers that need discarded entries

Common Pitfalls

- ⚠️ **Snapshotting without proper isolation**: Taking a snapshot while the state machine is changing can create an inconsistent snapshot. Solution: Use copy-on-write techniques or quiesce state machine operations during snapshot creation.
- ⚠️ **Discarding unapplied entries**: Accidentally discarding log entries that haven't been applied to the state machine yet. Solution: Only discard entries up to `lastApplied`, not `commitIndex`.
- ⚠️ **Incomplete snapshot metadata**: Forgetting to include the last included term breaks log consistency checks. Solution: Always store both `lastIncludedIndex` and `lastIncludedTerm` in the snapshot.

13.1.2 Leader Lease Reads for Linearizable Client Interactions

Mental Model: The Concert Ticket Validation System

Imagine a concert venue with multiple entry gates, each with its own record of ticket validity. To prevent ticket forgery, the gates periodically check with a central authority (the leader) about the latest security codes. However, constantly checking would create long lines. Instead, the central authority gives each gate a **lease**—a promise that "for the next 30 seconds, I guarantee no new security codes will be issued." During this lease period, gates can validate tickets independently without contacting the authority, ensuring fast entry while maintaining security. This is the essence of leader lease reads in Raft: trading some staleness for dramatically reduced read latency.

Concept and Rationale

Raft provides **linearizability** for reads (each read returns the result of the most recent write) by default through the leader: the leader checks it's still the leader (via heartbeat responses from a quorum) before serving a read. This ensures the read isn't stale but adds significant latency (one round trip to a quorum). **Leader lease** optimizes this by allowing the leader to serve reads without quorum confirmation for a bounded time period, under the guarantee that no other leader could have been elected during that period.

The core insight is that election timeouts (typically 150-300ms) create a natural "lease" period: if the current leader has recently received acknowledgments from a quorum, it knows no other node could become leader until at least one election timeout passes (since followers reset their timers on hearing from the leader).

Key Design Decisions

> **Decision: Lease Duration Calculation**
>
> - **Context**: Determining how long a leader can safely serve reads without quorum confirmation.
> - **Options Considered**:
>   1. **Fixed fraction of election timeout**: e.g., lease = election_timeout / 2
>   2. **Based on last heartbeat timestamps**: Track when quorum last acknowledged leadership
>   3. **Clock-based with bounded drift**: Use synchronized clocks with maximum drift allowance
> - **Decision**: Use clock-based lease with bounded clock drift assumption
> - **Rationale**: Clock-based leases are simpler to implement and reason about than heartbeat-based calculations. Modern data centers have clock synchronization (NTP) with drift typically <10ms, which is acceptable for election timeouts in the hundreds of milliseconds.
> - **Consequences**: Requires reasonably synchronized clocks and careful handling of clock jumps. Adds dependency on system clock stability.

Option	Pros	Cons	Chosen?
Clock-based lease	Simple implementation, clear expiration semantics	Depends on clock synchronization, vulnerable to clock jumps	**Yes**
Heartbeat-based	Pure Raft protocol, no clock assumptions	Complex to calculate, depends on network timing	No
Fixed timeout	Extremely simple	Overly conservative, wastes available lease time	No

Implementation Considerations

1. **Lease granularity**: Typically apply lease at the leader level (all reads benefit) rather than per-read
2. **Lease renewal**: The lease is implicitly renewed with each successful heartbeat quorum
3. **Clock safety margins**: Account for maximum expected clock drift (e.g., lease = election_timeout - 2*max_clock_drift)
4. **Read index optimization**: Combine with **read index** protocol for safety without log writes
5. **Follower reads**: Extend to allow followers to serve stale reads for use cases that tolerate eventual consistency

Common Pitfalls

- ⚠️ **Ignoring clock jumps**: System clock adjustments (NTP corrections) can invalidate lease calculations. Solution: Use monotonic clocks for lease duration calculations, not wall clocks.
- ⚠️ **Overly aggressive leases**: Setting lease duration too close to election timeout risks serving stale reads during leader changes. Solution: Conservative margins (e.g., lease ≤ election_timeout/2).
- ⚠️ **Not handling lease expiration**: Continuing to serve reads after lease expiration breaks linearizability. Solution: Track lease expiration timestamp and check before each read.

13.1.3 Production-Grade Transport Layer

Mental Model: The Diplomatic Courier Service

Imagine ambassadors communicating through a basic postal service—letters sometimes get lost, arrive out of order, or are delayed for days. A professional diplomatic courier service adds reliability layers: numbered dispatches with acknowledgments, batch deliveries for efficiency, encrypted channels for security, and priority routing for urgent messages. Upgrading Raft's transport is similar: transforming basic RPCs into a robust communication substrate that handles real-world network pathologies efficiently.

Concept and Rationale

The educational implementation likely uses simple HTTP/JSON RPCs or an in-memory transport for testing. Production systems require:

1. **Efficiency**: Serialization overhead (JSON) and connection establishment (HTTP/1.1) are significant
2. **Pipelineing**: Concurrent RPCs without waiting for responses
3. **Flow control**: Prevent fast leaders from overwhelming slow followers
4. **Connection management**: Persistent connections, reconnection strategies, load balancing
5. **Observability**: Metrics for RPC latency, throughput, error rates

Implementation Approaches

Transport Option	Protocol	Serialization	Best For	Implementation Complexity
gRPC	HTTP/2	Protocol Buffers	High-performance production systems	Medium-High
Custom TCP	Raw TCP	Custom binary format	Maximum performance, control	High
HTTP/2 + JSON	HTTP/2	JSON	Balance of simplicity and performance	Medium
HTTP/1.1 + JSON	HTTP/1.1	JSON	Simple implementation, debugging ease	Low

Key Components to Implement

1. **Connection pooling**: Reuse connections to avoid TCP handshake overhead
2. **Request batching**: Combine multiple AppendEntries for same follower into single RPC
3. **Compression**: Apply snappy/gzip compression for large snapshots or batched entries
4. **Backpressure**: Implement window-based flow control
5. **Metrics integration**: Track RPC counts, latencies, errors per peer

Common Pitfalls

- ⚠️ **Head-of-line blocking**: Using HTTP/1.1 with single connection causes sequential request processing. Solution: Use HTTP/2 or multiple parallel connections.
- ⚠️ **Silent connection drops**: TCP connections can die without FIN packets. Solution: Implement application-level keepalives/heartbeats.

- ⚠️ **Unbounded memory growth**: Queuing unlimited outgoing RPCs during network partitions. Solution: Implement bounded queues with backpressure.

13.1.4 Integration with a Key-Value Store

Mental Model: The Distributed Filing Cabinet

Imagine a traditional filing cabinet that only one person can access at a time. To scale to an entire office, you create a distributed version: multiple cabinets with synchronized contents, where a receptionist (the Raft leader) coordinates all file operations, ensuring everyone sees the same file versions regardless of which cabinet they check. This transforms Raft from an abstract consensus layer into a concrete, usable storage system.

Concept and Rationale

Raft's value is realized when paired with a state machine. A key-value store is the canonical example because:

1. It's universally useful (caching, configuration storage, service discovery)
2. It demonstrates all Raft features (reads, writes, linearizability)
3. It's simple enough for an extension project but complete enough to be practical

The integration involves:

- Defining key-value operations as commands in the Raft log
- Implementing a key-value state machine that applies these commands
- Adding client API (GET, PUT, DELETE) that proposals to Raft
- Supporting transactions or compare-and-swap operations

Architecture Pattern

Advanced Features for KV Store

1. **Session consistency**: Client sessions with sequence numbers for exactly-once semantics
2. **Range queries**: Efficient scanning of key ranges with snapshot isolation
3. **TTL/expiration**: Time-to-live for keys with background cleanup
4. **Watch/notify**: Client notifications for key changes
5. **Multi-version concurrency control**: Keeping multiple versions for snapshot reads

13.1.5 Dynamic Membership with Learner Nodes

Mental Model: The Corporate Boardroom with Apprentices

Imagine a corporate board that makes decisions by majority vote. Adding new board members requires careful onboarding: first they attend as **observers** (learners), watching discussions but not voting. Once they've caught up on all past decisions, they're promoted to full voting members. Similarly, removing members happens in two stages: first they lose voting rights but remain informed, then they're fully removed. This minimizes disruption during membership changes.

Concept and Rationale

The basic joint consensus implementation (Milestone 4) handles membership changes but has limitations:

1. **Availability impact**: The cluster is vulnerable during configuration changes
2. **New node impact**: Adding a slow node immediately affects quorum size
3. **No observer role**: All members are voting members

Learner nodes (non-voting members) address these issues by:

- Allowing new nodes to catch up log entries before joining the voting set
- Enabling read scalability (learners can serve reads)
- Providing a "buffer" for node replacement without affecting quorum size

Implementation Approach

1. **Extended configuration**: Add `learners` set to `ConfigState`
2. **Log replication to learners**: Leaders replicate to learners but don't count them for quorum
3. **Promotion protocol**: Learner → voting member via configuration change
4. **Client redirection**: Clients can read from learners for eventually consistent reads

Configuration State with Learners

```go
type ConfigState struct {
    Current    Configuration  // Voting members
    Pending    *Configuration // Pending configuration (if in joint consensus)
    Learners   map[int]bool   // Learner node IDs
    // Learners don't vote but receive log replication
}
```

13.1.6 Disk-Based Persistent Storage

Mental Model: The Court Reporter's Notebook and Archive

Imagine a court reporter taking notes in a small notebook during proceedings, then transcribing them each evening into a permanent archive. The notebook provides fast, temporary storage during active work, while the archive ensures no record is ever lost. Similarly, a production Raft implementation needs both in-memory structures for performance and durable disk storage for crash recovery.

Concept and Rationale

The educational implementation likely uses in-memory storage (`MemoryStore`) for simplicity. Production systems require:

1. **Crash recovery**: State must survive power loss and crashes
2. **Performance**: Efficient serialization and disk I/O patterns
3. **Consistency**: Guarantees that persisted state matches in-memory state

Storage Hierarchy Design

Storage Level	Purpose	Performance	Durability
In-memory structures	Active Raft state (nextIndex, matchIndex)	Nanosecond access	Volatile
Write-ahead log (WAL)	Log entries, votedFor, currentTerm	Microsecond append	Durable (fsync)
Snapshots	Periodic state machine checkpoints	Seconds to write/read	Durable
Metadata file	Current term, votedFor (for quick restart)	Microsecond update	Durable

Implementation Recommendations

1. **Use an embedded database**: RocksDB, LevelDB, or Badger provide efficient key-value storage with persistence
2. **Separate WAL and state storage**: WAL for log entries, KV store for Raft metadata and snapshots
3. **Batch fsync operations**: Group multiple writes before calling fsync() for better throughput
4. **Corruption detection**: Add checksums to all disk structures
5. **Recovery procedure**: Validate and rebuild state from WAL and latest snapshot on restart

13.1.7 Monitoring and Operational Tooling

Mental Model: The Air Traffic Control Dashboard

Imagine pilots flying blind with no instruments or ground communication—dangerous and inefficient. An air traffic control dashboard provides real-time visibility: aircraft positions, weather conditions, runway status. Similarly, operating a Raft cluster without monitoring is flying blind. Comprehensive observability transforms the system from a black box into a manageable service.

Key Monitoring Dimensions

Dimension	What to Monitor	Why It Matters
Raft Metrics	Term changes, leader elections, commit latency	Detect instability, performance issues
Log Health	Log size, unapplied entries, snapshot age	Prevent storage exhaustion, catch replication lag
Network	RPC latency, error rates, partition detection	Identify network issues affecting consensus
Resource Usage	Memory, CPU, disk I/O	Capacity planning, anomaly detection
Business Logic	State machine operations, client request rates	Application-specific health

Implementation Approach

1. **Structured logging**: Enhance the `Logger` to output machine-readable logs (JSON)
2. **Metrics collection**: Integrate with Prometheus or OpenTelemetry
3. **Health endpoints**: HTTP endpoints for readiness/liveness probes
4. **Admin API**: Operations like manual leader transfer, configuration changes
5. **Dashboard**: Real-time visualization of cluster state

Example Monitoring Integration

```go
// Enhanced RaftNode with metrics
type InstrumentedRaftNode struct {
    *RaftNode
    metrics *RaftMetrics
}

type RaftMetrics struct {
    TermGauge          prometheus.Gauge
    ElectionCounter    prometheus.Counter
    CommitLatencyHist  prometheus.Histogram
    RPCErrorCounter    *prometheus.CounterVec // by peer and type
    LogSizeGauge       prometheus.Gauge
    // ... additional metrics
}
```

Implementation Guidance

While the future extensions are advanced topics, here are starting points for implementing the most critical one: log compaction and snapshotting.

Technology Recommendations

Component	Simple Option	Advanced Option
Snapshot Storage	Single file on disk	Distributed object store (S3-compatible)
Snapshot Format	Go's `gob` encoding	Protocol Buffers with compression
Snapshot Transfer	HTTP with range requests	gRPC streaming
State Machine Isolation	Stop-the-world copy	Copy-on-write with fork()

Recommended File Structure

```
raft/
├── snapshot/
│   ├── snapshotter.go      # Snapshot creation and management
│   ├── storage.go          # Snapshot storage backend interface
│   ├── file_storage.go     # Disk-based snapshot storage
│   └── install_snapshot.go # InstallSnapshot RPC handler
├── statemachine/
│   └── kv_store.go         # Key-value store with snapshot support
└── transport/
    └── snapshot_stream.go  # Snapshot streaming over HTTP/2
```

Infrastructure Starter Code: Snapshot Storage Interface

GO

```go
// raft/snapshot/storage.go

package snapshot

import (
    "io"
)

// SnapshotMetadata contains Raft consistency information for a snapshot
type SnapshotMetadata struct {
    LastIncludedIndex int
    LastIncludedTerm  int
    Configuration     Configuration
    ConfigurationIdx  int
}

// Snapshot represents a point-in-time capture of the state machine
type Snapshot struct {
    Metadata SnapshotMetadata
    Data     []byte  // Serialized state machine state
    CRC32    uint32  // Checksum for integrity verification
}

// Storage defines the interface for snapshot persistence
type Storage interface {
    // CreateSnapshot persists a snapshot with given metadata and data
    CreateSnapshot(metadata SnapshotMetadata, data []byte) (string, error)

    // GetSnapshot retrieves a snapshot by ID
    GetSnapshot(id string) (*Snapshot, error)

    // ListSnapshots returns all available snapshots sorted by index (newest first)
    ListSnapshots() ([]SnapshotMetadata, error)

    // DeleteSnapshot removes a snapshot
    DeleteSnapshot(id string) error

    // LatestSnapshot returns the most recent snapshot metadata
    LatestSnapshot() (*SnapshotMetadata, error)
```

```go
}

// Snapshotter manages snapshot creation and installation
type Snapshotter struct {
    storage Storage
    maxLogSize int // Trigger snapshot when log exceeds this size
    snapshotInProgress bool
    mu sync.RWMutex
}

func NewSnapshotter(storage Storage, maxLogSize int) *Snapshotter {
    return &Snapshotter{
        storage: storage,
        maxLogSize: maxLogSize,
    }
}

// ShouldSnapshot returns true if snapshot should be created
func (s *Snapshotter) ShouldSnapshot(logSize int, lastSnapshotIdx int) bool {
    s.mu.RLock()
    defer s.mu.RUnlock()

    // Don't create new snapshot if one is in progress
    if s.snapshotInProgress {
        return false
    }

    // Snapshot if log size exceeds threshold
    return logSize > s.maxLogSize
}

// BeginSnapshot marks that snapshot creation has started
func (s *Snapshotter) BeginSnapshot() error {
    s.mu.Lock()
    defer s.mu.Unlock()

    if s.snapshotInProgress {
        return errors.New("snapshot already in progress")
```

```
    }


    s.snapshotInProgress = true
    return nil
}

// EndSnapshot marks that snapshot creation has completed
func (s *Snapshotter) EndSnapshot() {
    s.mu.Lock()
    defer s.mu.Unlock()
    s.snapshotInProgress = false
}
```

Core Logic Skeleton: InstallSnapshot RPC Handler

```go
// raft/snapshot/install_snapshot.go

package snapshot

import (
    "fmt"
    "io"
)

// InstallSnapshotArgs is the RPC arguments for installing a snapshot
type InstallSnapshotArgs struct {
    Term              int    // Leader's term
    LeaderID          int    // Leader ID for redirects
    LastIncludedIndex int    // Snapshot replaces all entries up to this index
    LastIncludedTerm  int    // Term of LastIncludedIndex
    Offset            int    // Byte offset where data is placed in snapshot file
    Data              []byte // Raw snapshot data chunk
    Done              bool   // True if this is the last chunk
}

// InstallSnapshotReply is the RPC response
type InstallSnapshotReply struct {
    Term int // Current term for leader to update itself
}

// handleInstallSnapshot processes incoming snapshot installation requests
func (rn *RaftNode) handleInstallSnapshot(args InstallSnapshotArgs) InstallSnapshotReply {
    // TODO 1: Reply immediately if term < currentTerm (Raft paper §7)


    // TODO 2: Check if this is the first chunk (offset == 0)
    //         If yes, create new temporary snapshot file


    // TODO 3: Write data to snapshot file at specified offset


    // TODO 4: If this is the last chunk (Done == true):
    //         a. Verify snapshot integrity (checksum)
    //         b. Save snapshot to persistent storage
    //         c. Apply snapshot to state machine
    //         d. Compact log (discard entries up to LastIncludedIndex)
```

```
    //        e. Update commitIndex and lastApplied if needed
    //        f. Delete temporary file

    // TODO 5: Return current term so leader can update itself if stale

    return InstallSnapshotReply{Term: rn.currentTerm}
}

// sendInstallSnapshot sends snapshot chunks to a follower
func (rn *RaftNode) sendInstallSnapshot(peerID int, snapshotID string) error {
    // TODO 1: Load snapshot from storage

    // TODO 2: Split snapshot data into chunks (e.g., 64KB each)

    // TODO 3: For each chunk, send InstallSnapshot RPC with appropriate offset

    // TODO 4: Handle retries for failed chunks with exponential backoff

    // TODO 5: If follower's term is higher, step down to follower

    // TODO 6: Update follower's matchIndex and nextIndex after successful installation

    return nil
}
```

Language-Specific Hints for Go

1. **Snapshot creation**: Use `encoding/gob` for simple serialization or Protocol Buffers for efficiency
2. **File handling**: Use `os.File` with `Sync()` for durability, `io.CopyBuffer` for efficient chunking
3. **Concurrent access**: Use `sync.RWMutex` to allow reads during snapshot creation
4. **Background snapshotting**: Use a goroutine with context for cancellation:

```go
go func(ctx context.Context) {
    select {
    case <-ctx.Done():
        return
    case <-snapshotTrigger:
        createSnapshot()
    }
}(ctx)
```

5. **Checksums**: Use `hash/crc32` for integrity verification of snapshot data

Debugging Tips for Snapshotting

Symptom	Likely Cause	How to Diagnose	Fix
State machine divergence after snapshot	Snapshot taken during state machine update	Check if snapshot creation isolates state machine	Implement copy-on-write or quiesce operations during snapshot
High memory usage	Too many snapshots retained	Check snapshot retention policy	Implement automatic cleanup of old snapshots
Snapshot transfer stalls	Large snapshot without chunking	Monitor network transfer progress	Implement chunking with progress tracking and resume capability
Log entries disappear	Log compaction deletes unapplied entries	Compare `lastApplied` with snapshot index	Only compact up to `lastApplied`, not `commitIndex`

14. Glossary

> **Milestone(s):** 1 (Leader Election), 2 (Log Replication), 3 (Safety Properties), 4 (Cluster Membership Changes)

This glossary provides definitive explanations for all technical terms, acronyms, and domain-specific vocabulary used throughout this design document. It serves as a quick reference to reinforce understanding and ensure consistent terminology across the implementation effort.

14.1 Term Definitions

Term	Definition	Section Reference
AppendEntries RPC	The primary RPC used by a Raft leader to replicate log entries to followers and to send heartbeats. It includes fields for leader term, previous log index/term for consistency checking, new entries, and the leader's commit index.	Section 4.3 (Data Model), Section 6 (Log Replication)
CAP Theorem	The fundamental trade-off in distributed systems between Consistency (all nodes see the same data), Availability (every request receives a response), and Partition tolerance (the system continues operating despite network splits). Raft is designed as a CP (Consistent and Partition-tolerant) system.	Section 1.2 (Why Consensus is Difficult)
Candidate	One of the three Raft node states. A node transitions to Candidate when its election timeout expires. In this state, it increments its current term, votes for itself, and issues `RequestVote` RPCs to other nodes in an attempt to become the leader for that term.	Section 5 (Leader Election), Naming Conventions
Catch-up	The process by which a new or lagging server (including a **Learner**) replicates missing log entries from the leader before it is allowed to participate fully in the cluster (e.g., vote in elections). This ensures the **Leader Completeness** property.	Section 8 (Cluster Membership Changes)
Chain of Custody	A mental model for Raft safety, analogous to legal evidence handling, where every transfer (log entry replication) is logged and verified to prevent alteration or loss, ensuring a tamper-proof sequence of commands.	Section 7.1 (Mental Model: The Chain of Custody)
Commit Index	The highest index of a log entry known to be **committed**. The leader advances this index when an entry has been stored on a majority of servers. All servers eventually apply entries up to their commit index to their **State Machine**.	Section 4.2 (Volatile State), Section 6 (Log Replication)
Committed Entry	A log entry that is considered permanent and will be applied by all non-failed nodes. An entry is committed once the leader has replicated it to a majority of the cluster (`commitIndex` is updated) and will survive future leader changes.	Section 6 (Log Replication)
Configuration	The set of servers that are voting members in the Raft cluster. This is changed via special log entries and the **Joint Consensus** protocol. Represented by the `Configuration` struct.	Section 8 (Cluster Membership Changes), Naming Conventions
Consensus	The process of achieving agreement among a set of distributed processes on a single value or, in Raft's case, an ordered sequence of values (log entries).	Section 1 (Context and Problem Statement)
Consistency Check	The verification performed by a follower when processing an `AppendEntriesArgs` RPC. The follower checks that its log contains an entry at `PrevLogIndex` with a term matching `PrevLogTerm`. If not, it rejects the RPC, triggering **Log Reconciliation**.	Section 6 (Log Replication)
Crash-stop	A failure model where servers fail by stopping (crashing) and later may restart, but do not exhibit arbitrary or malicious (**Byzantine**) behavior. Raft is designed to tolerate crash-stop failures.	Section 10.1 (Failure Modes and Detection)
Deterministic Simulation	A testing approach that controls randomness (e.g., election timeouts) and timing to make distributed system behavior reproducible, enabling the debugging of complex failure scenarios.	Section 11.2 (Property-Based Testing), Section 12 (Debugging Guide)
Disjoint Majorities	A dangerous scenario during cluster membership changes where the old configuration and new configuration have non-overlapping majorities, which could potentially lead to two leaders being elected. The **Joint Consensus** protocol prevents this.	Section 8.3 (Common Pitfalls in Membership Changes)
Election Safety	A core Raft safety property guaranteeing that at most one leader can be elected for any given **Term**. This is ensured by the rule that a node grants a vote only if it has not already voted for another candidate in the same term.	Section 7 (Safety Properties)
Event Trace	A recorded, chronological sequence of events (e.g., RPCs, state transitions) across all nodes in a cluster, used for post-mortem analysis and debugging test failures.	Section 12.2 (Debugging Techniques)

Term	Definition	Section Reference
Figure 8 Scenario	A specific safety violation scenario from the Raft paper (Figure 8) where a leader could incorrectly commit an entry from a previous term if it replicates it to a majority but then crashes before committing it. The safety rule (a leader only commits entries from its own term) prevents this.	Section 7.3 (Common Pitfalls), Naming Conventions
Follower	One of the three Raft node states. A node starts as a Follower, responding to RPCs from candidates and leaders. If it does not receive communication from a valid leader within its election timeout, it transitions to **Candidate**.	Section 5 (Leader Election), Naming Conventions
gRPC	A high-performance RPC (Remote Procedure Call) framework using HTTP/2 and Protocol Buffers. Mentioned as an advanced option for the **Transport** layer.	Section 13 (Future Extensions), Naming Conventions
Heartbeat	An empty `AppendEntries` RPC (with no log entries) sent by the leader to all followers. Its primary purpose is to maintain authority and prevent followers from starting new elections.	Section 6 (Log Replication)
Heisenbug	A class of bug that disappears or alters its behavior when one attempts to study or debug it, often due to timing changes introduced by adding logging or instrumentation.	Section 12.2 (Debugging Techniques)
InstrumentedRaftNode	A wrapper around the core `RaftNode` that attaches metrics collection (e.g., gauges, counters) for **Observability**.	Naming Conventions
Invariant Checking	The runtime verification of **Safety Invariants** during tests. Code is added to assert that certain properties (e.g., "no two leaders in the same term") always hold.	Section 11.2 (Property-Based Testing)
Joint Consensus	The protocol used by Raft for safe cluster membership changes. During a transition, decisions require majority agreement from *both* the old configuration and the new configuration, preventing **Disjoint Majorities**.	Section 8 (Cluster Membership Changes), Naming Conventions
Last Applied Index	The index of the highest log entry that has been applied to the node's **State Machine**. This is a volatile state, always less than or equal to the **Commit Index**.	Section 4.2 (Volatile State)
Leader	One of the three Raft node states. The authoritative node that coordinates all operations. It accepts client commands, replicates them to followers via `AppendEntries` RPCs, and determines when entries are **committed**. Only one leader can exist per **Term**.	Section 5 (Leader Election), Naming Conventions
Leader Completeness	A core Raft safety property guaranteeing that any leader elected for a given term contains all entries that were **committed** in previous terms. This is ensured by the **Log Comparison** rule during voting.	Section 7 (Safety Properties)
Leader Lease	An optional optimization (not part of core Raft) where a leader can serve linearizable reads without contacting a quorum, based on the assumption it remains leader for a known period of time.	Section 13 (Future Extensions), Naming Conventions
Leader Step-Down	The transition from the **Leader** state to the **Follower** state, which occurs when a leader discovers (via an RPC response or request) that there exists a node with a higher **Term**.	Section 10.2 (Recovery and State Reconciliation)
Learner / Learner Node	A non-voting Raft member that receives log replication but does not participate in elections or count toward majorities. Used to safely introduce new nodes to the cluster, allowing them to **Catch-up** before becoming full voting members.	Section 8 (Cluster Membership Changes), Naming Conventions
Linearizability	A strong consistency guarantee for concurrent systems where each operation (read or write) appears to take effect atomically at some point between its invocation and response. A system with linearizable reads will always return the result of the most recent write.	Section 13 (Future Extensions), Naming Conventions
Log	An ordered sequence of **Log Entry** structures stored by each Raft node. It is the primary vehicle for achieving consensus.	Section 4.1 (Persistent State)
Log Compaction	The process of discarding old log entries while preserving the system's state, typically by taking a **Snapshot** of the **State Machine**. This bounds storage growth.	Section 13 (Future Extensions), Naming Conventions

Term	Definition	Section Reference
Log Comparison	The process of determining which of two logs is "more up-to-date" during a `RequestVote` RPC. Comparison is first by the term of the last log entry, and then by the index of the last log entry if terms are equal.	Section 5 (Leader Election)
Log Entry	The fundamental unit of work in Raft, consisting of a **Term** number, an **Index** (position in the log), and a **Command** (the application data). Represented by the `LogEntry` struct.	Section 4.1 (Persistent State), Naming Conventions
Log Matching Property	A fundamental Raft invariant: if two logs contain an entry with the same index and term, then the logs are identical in all entries up to that index. This property is maintained by the **Consistency Check** in `AppendEntries`.	Section 7 (Safety Properties), Naming Conventions
Log Reconciliation	The process by which a leader, upon receiving a rejection from a follower's **Consistency Check**, decrements the `nextIndex` for that follower and retries the `AppendEntries` RPC to find the point where the logs match.	Section 6 (Log Replication)
Log Store	The abstracted component responsible for storing and retrieving **Log Entry** objects. In the educational implementation, this is often an in-memory structure like `MemoryStore`.	Section 3.1 (Component Overview), Naming Conventions
LogEvent	A structured log entry used for **Observability**, containing fields like timestamp, node ID, term, state, and event details.	Naming Conventions
Liveness Properties	System guarantees that something good *eventually* happens (e.g., a leader will eventually be elected, a submitted command will eventually be committed). Contrast with **Safety Invariants**.	Section 11.2 (Property-Based Testing), Naming Conventions
Majority (Quorum)	More than half of the voting members in the current cluster **Configuration**. Many Raft operations (electing a leader, committing an entry) require agreement from a majority.	Section 1.1 (The Replicated State Machine Analogy)
Match Index	A leader's volatile state (per follower) tracking the highest known log index that has been successfully replicated to that follower. Used to calculate the **Commit Index**.	Section 4.2 (Volatile State)
Network Partition	A network failure that splits the cluster into two or more groups that cannot communicate with each other. Raft handles partitions by ensuring only the partition with a **Majority** of servers can elect a leader and make progress.	Section 10.1 (Failure Modes and Detection), Naming Conventions
Next Index	A leader's volatile state (per follower) tracking the index of the next log entry to send to that follower. Initialized to the leader's last log index + 1, and adjusted downward during **Log Reconciliation**.	Section 4.2 (Volatile State)
Observability	The comprehensive monitoring, logging, and tracing of system behavior to understand its internal state and diagnose problems. Implemented via structured **LogEvent** recording and metrics collection.	Section 12.2 (Debugging Techniques), Naming Conventions
Paxos	A family of consensus protocols predating Raft, known for being difficult to understand and implement correctly. Raft was designed to be more understandable while providing equivalent safety guarantees.	Section 1.3 (Pre-Raft Landscape)
Persistent State	The portion of a Raft node's state that must be written to stable storage (e.g., disk) before responding to certain RPCs. It includes `currentTerm`, `votedFor`, and the `log`. This state survives crashes.	Section 4.1 (Persistent State), Naming Conventions
Property-Based Testing	A testing approach that generates random inputs and sequences of operations to verify that system **Invariants** hold under all conditions, rather than testing specific examples.	Section 11.2 (Property-Based Testing), Naming Conventions
Quorum	See **Majority**.	Naming Conventions
Quorum Intersection	The mathematical property that any two majorities (quorums) in a set must overlap by at least one member. This property underpins Raft's safety guarantees, ensuring that decisions made in one term are known to at least one node in any subsequent majority.	Section 7 (Safety Properties), Naming Conventions

Term	Definition	Section Reference
Raft	A consensus algorithm for managing a **Replicated State Machine**. It is designed to be understandable and equivalent to Paxos in fault-tolerance and performance.	Section 1 (Context and Problem Statement)
Randomized Fault Injection	A testing technique that intentionally introduces random network partitions, message loss, and server crashes to validate system robustness and recovery mechanisms.	Section 11.2 (Property-Based Testing), Naming Conventions
Randomized Timeout	An election timeout duration chosen randomly from a configured range for each node. This reduces the likelihood of **Split Votes** by making it less probable that multiple nodes will start elections simultaneously.	Section 5 (Leader Election), Naming Conventions
Replicated State Machine	A fundamental approach to building fault-tolerant services where multiple servers start from the same initial state and apply the same sequence of deterministic commands, thus maintaining identical state. Raft is a protocol to achieve this.	Section 1.1 (The Replicated State Machine Analogy), Naming Conventions
RequestVote RPC	The RPC used by **Candidate** nodes to solicit votes during leader elections. It includes the candidate's term and information about its log (**Log Comparison**) to ensure **Leader Completeness**.	Section 5 (Leader Election), Section 4.3 (Data Model)
Safety Invariants	Properties that must *always* hold for a system to be correct (e.g., "no two leaders in the same term", "all nodes apply the same commands in the same order"). Contrast with **Liveness Properties**.	Section 11.2 (Property-Based Testing), Naming Conventions
Safety Properties	The collective correctness guarantees of the Raft protocol, primarily **Election Safety**, **Leader Completeness**, and **State Machine Safety**.	Section 7 (Safety Properties), Naming Conventions
Snapshot	A point-in-time capture of the **State Machine**'s state, accompanied by Raft metadata (last included index/term, configuration). Used for **Log Compaction**. Represented by the `Snapshot` struct.	Section 13 (Future Extensions), Naming Conventions
Split Vote	An election scenario where two or more **Candidate** nodes receive votes, but none achieves a **Majority**, causing the election to fail and a new one to be started (after new randomized timeouts).	Section 5.2 (ADR: Randomized Election Timeouts)
State Inspector	A debugging tool that exposes the internal state of Raft nodes (e.g., term, role, log summary) via an API or HTTP endpoint, allowing developers to inspect cluster health without intrusive logging.	Section 12.2 (Debugging Techniques), Naming Conventions
State Machine	The application-specific logic that applies committed commands from the Raft log to produce some external effect (e.g., modifying a key-value store). Represented by the `StateMachine` interface with an `Apply` method.	Section 3.1 (Component Overview), Naming Conventions
State Machine Safety	A core Raft safety property guaranteeing that if a server has applied a log entry at a given index to its state machine, no other server will ever apply a different log entry for the same index.	Section 7 (Safety Properties)
Structured Logging	A logging approach where each log entry follows a predefined format with consistent fields (e.g., timestamp, level, event, details), making logs machine-parseable and suitable for automated analysis.	Section 12.2 (Debugging Techniques), Naming Conventions
Term	A monotonically increasing epoch number that identifies a leader's reign. Each node tracks the `currentTerm` in persistent storage. Terms are used to detect stale information (e.g., from a former leader) and ensure **Election Safety**.	Section 4.1 (Persistent State), Naming Conventions
Test Harness	The infrastructure (code and scripts) used to start a cluster of Raft nodes, inject commands and failures, and observe the resulting behavior during automated testing.	Section 11.2 (Property-Based Testing), Naming Conventions
Trace Visualization	Creating a graphical timeline of **Event Traces** across nodes to intuitively understand the sequence of RPCs, state transitions, and other events during a test or operation.	Section 12.2 (Debugging Techniques), Naming Conventions

Term	Definition	Section Reference
Transport	The abstracted network communication layer responsible for delivering RPCs (`RequestVote` , `AppendEntries`) between Raft nodes. Implementations can be simple HTTP (`HTTPTransport`) or a simulated network (`VirtualTransport`).	Section 3.1 (Component Overview), Naming Conventions
Two-Phase Commit Pattern	A high-level description of Raft's normal operation for log replication: Phase 1 (Replication) where the leader sends entries to all followers, and Phase 2 (Commit) where the leader advances the commit index after a majority acknowledge.	Section 6.1 (Mental Model), Naming Conventions
Virtual Transport	A simulated network transport layer used for deterministic testing. It can model message delays, drops, and network partitions without real network sockets.	Section 11.2 (Property-Based Testing), Naming Conventions
Volatile State	The portion of a Raft node's state that is reinitialized after a restart. It includes `commitIndex` , `lastApplied` , `nextIndex[]` , and `matchIndex[]` (for leaders). This state can be reconstructed from persistent state and RPCs.	Section 4.2 (Volatile State), Naming Conventions
Write-ahead Log (WAL)	A persistent log where operations are recorded before they are applied to the state machine, ensuring durability. In Raft, the persistent **log** itself serves as the WAL.	Section 13 (Future Extensions), Naming Conventions