

Distributed Consensus (Raft/Paxos)

1. The Real Problem This Exists to Solve

In distributed systems, multiple nodes must reach agreement on state despite unreliable networks, node failures, and arbitrary delays. Consensus algorithms provide safety guarantees that ensure all nodes converge to the same decision and never diverge, even under the worst conditions.

Real production scenario:

- Distributed database cluster needs to elect leader reliably
- **Without proper consensus algorithm:**
 - Manual election: Pick node 1 as primary
 - Node 1 fails → Manual intervention required to pick new primary
 - Takes 30 minutes to detect failure, choose new primary, update config
 - User traffic diverted during 30 minutes
 - During failover: Old primary might reconnect, causing conflicts
 - **Result: 30 minute outages, manual intervention, data divergence possible**
- **With Raft consensus:**
 - Cluster of 5 nodes
 - Automatic leader election (takes 150-300ms)
 - Network partition detected: 5ms
 - New leader elected: 300ms
 - Clients failover: Automatic retries
 - **Total downtime: <500ms, fully automatic, no manual intervention**

The fundamental problem: Achieving agreement in distributed systems is harder than it seems:

- Network messages can be delayed, reordered, or lost
- Nodes can fail at any time
- A node may fail while others think it's healthy (causing conflicting decisions)
- Simple majority voting doesn't handle all edge cases

Without proper consensus:

- Distributed systems break apart during failures
- Manual intervention required
- Data divergence possible
- No strong safety guarantees

With consensus algorithms:

- Automatic agreement despite failures
- Strong safety guarantees (no data divergence)
- Efficiency (minimal message overhead)
- Fault tolerance (survives minority failures)

2. The Naive / Incorrect Approaches (IMPORTANT)

✗ Incorrect Approach #1: Simple Majority Voting Without Term Numbers

```

// Incorrect: Voting without tracking epochs/terms
class SimpleVoting {
  private votes = new Map<string, number>(); // Node ID -> vote for which node
  private nodeIds = ['A', 'B', 'C', 'D', 'E'];

  async electLeader(): Promise<string | null> {
    // Ask all nodes who they think should be leader
    const votes = await this.collectVotes();

    // Count votes
    const tally = new Map<string, number>();
    for (const vote of votes) {
      tally.set(vote, (tally.get(vote) || 0) + 1);
    }

    // Find candidate with majority
    const majority = Math.floor(this.nodeIds.length / 2) + 1;
    for (const [candidate, count] of tally) {
      if (count >= majority) {
        return candidate; // Found leader!
      }
    }

    return null; // No consensus
  }

  private async collectVotes(): Promise<string[]> {
    // Get vote from each node (buggy: no term tracking)
    const votes: string[] = [];
    for (const nodeId of this.nodeIds) {
      const vote = await this.requestVote(nodeId);
      votes.push(vote);
    }
    return votes;
  }
}

// Problem scenario with 2 elections:
// Election 1 (nodes B, C, D):
//   B votes for B
//   C votes for B
//   D votes for B
//   Result: B elected with 3 votes
//
// Meanwhile during network delay, Election 2 (nodes A, E):
//   A votes for A
//   E votes for A
//   A elected with 2 votes (less than majority 3, but A thinks it's leader)
//
// Network heals: 2 leaders (A, B) from same election round!

```

Why it seems reasonable:

- Majority voting is mathematically sound
- Simpler than full consensus

How it breaks:

- No term/epoch numbers to distinguish election rounds
- Same quorum can vote in overlapping elections
- Two different leaders elected from same election (example above)
- Conflicting decisions possible when voting rounds overlap

Production symptoms:

- Two nodes claim to be leader at same time
- Data divergence during "two leader" period
- Clients don't know which leader to talk to
- No automatic recovery (two leaders persist)

✗ Incorrect Approach #2: Single Authoritative Server (Not Distributed Consensus)

```
// Incorrect: Centralized leader election
class CentralizedElection {
  private authorityServer = 'config-server.example.com';

  async electLeader(): Promise<string> {
    try {
      const response = await fetch(`${this.authorityServer}/leader`);
      return response.json().leader;
    } catch {
      throw new Error('Cannot reach authority server');
    }
  }
}

// Problem: If authority server down, no new leader can be elected
// Cluster paralyzed without external system
```

Why it seems reasonable:

- Simple to implement
- Single source of truth
- No complex distributed algorithm

How it breaks:

- Single point of failure
- If authority server down, entire cluster broken
- No recovery without external intervention
- Defeats purpose of distributed system

Production symptoms:

- Authority server becomes bottleneck
- Authority server down → entire cluster unavailable

- Worse availability than without redundancy

✗ Incorrect Approach #3: Last-Write-Wins Without Causality

```
// Incorrect: Assume later timestamp = correct winner
class LastWriteWins {
  private leaderClaim: { nodeId: string; timestamp: number } | null = null;

  async claimLeadership(nodeId: string): Promise<void> {
    const timestamp = Date.now();

    if (!this.leaderClaim || timestamp > this.leaderClaim.timestamp) {
      this.leaderClaim = { nodeId, timestamp };
      console.log(`${nodeId} is new leader (timestamp: ${timestamp})`);
    }
  }
}

// Problem scenario (clock skew):
// Node A at T=100ms claims leadership: "A is leader"
// Node B at T=50ms claims leadership (clock ahead): "B is leader"
// B's timestamp > A's timestamp, so B wins
// But B's clock is wrong! A was actually elected first
//
// Both nodes think they're primary, data diverges
```

Why it seems reasonable:

- Intuitive: Later claim should win
- Simple algorithm

How it breaks:

- Doesn't account for clock skew
- Doesn't account for network delays
- Multiple leaders possible if clocks wrong
- No causality guarantees

Production symptoms:

- Nodes with wrong clocks win elections
- Multiple leaders with clock-skewed nodes
- Data divergence with no recovery path

✗ Incorrect Approach #4: Consensus Without Log Replication

```
// Incorrect: Decide on leader but not on data consistency
class LeaderElectionOnly {
  private leader: string | null = null;

  async electLeader(): Promise<string> {
    // Elect leader correctly
    const votes = await this.consensusVote();
```

```

const majority = Math.floor(5 / 2) + 1;

for (const [nodeId, votes] of votes) {
  if (votes >= majority) {
    this.leader = nodeId;
    return nodeId;
  }
}

// But no mechanism to ensure data replication!
// Leader elected, but followers don't know what data to replicate
// Uncommitted entries on old leader not copied to new leader
// Data loss possible
}

```

Why it seems reasonable:

- Focuses on leader election (consensus problem solved)
- Assumes data handling separate

How it breaks:

- Consensus for leader \neq Consensus for data
- Old leader may have uncommitted entries
- New leader elected but doesn't know about old entries
- Followers may have different states

Production symptoms:

- Leader elected correctly
- But different followers have different data
- New leader doesn't have all data from previous leader
- Inconsistent state across cluster
- Data loss when followers fail

3. Correct Mental Model (How It Actually Works)

Consensus algorithms (like Raft) solve the problem of getting multiple nodes to agree on a sequence of decisions (log entries) despite arbitrary failures. The key insight is combining three mechanisms: election, log replication, and safety.

The Three Sub-Problems

1. Leader Election
 - Elect exactly one leader per term
 - Guarantee: Only one leader per term
2. Log Replication
 - Leader replicates entries to followers
 - Guarantee: All nodes have identical logs (up to commit point)
3. Safety

- Committed entries never lost
- Committed entries never overwritten
- Guarantee: Durability and linearizability

The State Machine Model

All nodes maintain three things:

1. Log (persistent)
 - [Entry 1, Entry 2, Entry 3, ...]
 - Appended only
 - Never overwritten
2. State Machine (replicated)
 - Apply log entries to state machine
 - All nodes apply same entries in same order
 - Results in identical state
3. Metadata (persistent)
 - Current term
 - Current vote
 - Commit index

The Term Model (Unique for Raft)

Each election cycle is a "term"

Term 1: A elected leader, replicates log

Term 2: A fails, B elected leader, continues replication

Term 3: Network partition, C elected leader in minority, then deposed

Key property:

- Only one leader per term
- Term numbers always increasing
- Old term leaders rejected (term too low)

Example of safety:

Term 1: A (leader) writes entry X

Term 2: A fails, but followers have entry X in log

Term 2: B elected, replicates X to all (majority)

Term 2: A comes back, but term 2 > term 1

A is rejected as obsolete (term 1 < term 2)

The Commit Point (Critical for Durability)

Not all entries in leader's log are committed:

Leader: [E1, E2, E3, E4*] (E4 not replicated yet)

Follower 1: [E1, E2, E3]

Follower 2: [E1, E2, E3]

Follower 3: [E1, E2]

Follower 4: [E1, E2]

Entry E1-E3: Replicated to majority → COMMITTED

Entry E4: Only on leader → NOT COMMITTED

Safety guarantee: Committed entries are durable
(E1-E3 survive even if leader crashes)

Guarantee: Uncommitted entries can be overwritten
(E4 can be replaced if leader crashes before replicating)

4. Correct Design & Algorithm

Raft Algorithm Overview

Server states:

Follower: Receives commands from leader

Candidate: Tries to become leader

Leader: Accepts client requests, replicates to followers

On timeout (no heartbeat from leader):

Candidate increments term

Candidate votes for itself

Candidate requests votes from other nodes

If gets majority votes:

Becomes leader

Sends heartbeats to all followers

Followers accept entries from leader (if term \geq current term)

Commit point advances when:

Entry replicated to majority

Leader sends commit index to followers

Followers apply all entries up to commit index

Safety Rules

A node can only vote once per term

(Prevents multiple leaders in same term)

A node only votes for candidate if:

- Candidate term \geq node's term

- Candidate log is at least as recent as node's log

(Ensures newly elected leader has all committed entries)

An entry is committed when:

- Replicated to majority nodes

- Leader is in current term (not from old term)

(Ensures committed entries are durable)

A follower only accepts an entry if:

- It matches leader's view (term and index align)

(Prevents conflicting logs)

5. Full Production-Grade Implementation

```
interface LogEntry {
  term: number;
  index: number;
  command: any;
}

interface ServerState {
  state: 'follower' | 'candidate' | 'leader';
  currentTerm: number;
  votedFor: string | null;
  log: LogEntry[];
  commitIndex: number;
  lastApplied: number;
  leaderNextIndex?: Map<string, number>;
  leaderMatchIndex?: Map<string, number>;
}

class RaftNode {
  private nodeId: string;
  private peers: Map<string, RaftNode>;
  private state: ServerState;
  private electionTimeout: NodeJS.Timeout | null = null;
  private heartbeatInterval: NodeJS.Timeout | null = null;
  private readonly electionTimeoutMs = 150 + Math.random() * 150; // 150-300ms
  private readonly heartbeatIntervalMs = 50; // 50ms
  private stateMachine: Map<string, any> = new Map();

  constructor(nodeId: string, peers: Map<string, RaftNode>) {
    this.nodeId = nodeId;
    this.peers = peers;
    this.state = {
      state: 'follower',
      currentTerm: 0,
      votedFor: null,
      log: [],
      commitIndex: 0,
      lastApplied: 0,
    };
  };

  this.resetElectionTimer();
}
```

```

/**
 * Request vote RPC (candidate to other nodes)
 */
async requestVote(
  term: number,
  candidateId: string,
  lastLogIndex: number,
  lastLogTerm: number
): Promise<{ term: number; voteGranted: boolean }> {
  // Update term if needed
  if (term > this.state.currentTerm) {
    this.state.currentTerm = term;
    this.state.votedFor = null;
  }

  // Reject if candidate term is old
  if (term < this.state.currentTerm) {
    return { term: this.state.currentTerm, voteGranted: false };
  }

  // Reject if already voted in this term
  if (this.state.votedFor !== null && this.state.votedFor !== candidateId) {
    return { term: this.state.currentTerm, voteGranted: false };
  }

  // Check if candidate log is up-to-date
  const lastEntry = this.getLastLogEntry();
  if (
    lastLogTerm < lastEntry.term ||
    (lastLogTerm === lastEntry.term && lastLogIndex < lastEntry.index)
  ) {
    // Candidate log is behind
    return { term: this.state.currentTerm, voteGranted: false };
  }

  // Grant vote
  this.state.votedFor = candidateId;
  return { term: this.state.currentTerm, voteGranted: true };
}

/**
 * Append entries RPC (leader to followers)
 */
async appendEntries(
  term: number,
  leaderId: string,
  prevLogIndex: number,
  prevLogTerm: number,
  entries: LogEntry[],
  leaderCommit: number
): Promise<{ term: number; success: boolean }> {
  // Update term if needed

```

```

    if (term > this.state.currentTerm) {
        this.state.currentTerm = term;
        this.state.votedFor = null;
        this.state.state = 'follower'; // Step down if we're leader
    }

    // Reject if term is old
    if (term < this.state.currentTerm) {
        return { term: this.state.currentTerm, success: false };
    }

    // Confirm leader
    this.resetElectionTimer();

    // Check log consistency
    if (prevLogIndex > 0) {
        const prevEntry = this.state.log[prevLogIndex - 1];
        if (!prevEntry || prevEntry.term !== prevLogTerm) {
            return { term: this.state.currentTerm, success: false };
        }
    }

    // Delete conflicting entries
    if (entries.length > 0) {
        const firstNewIndex = prevLogIndex;
        for (let i = 0; i < entries.length; i++) {
            const index = firstNewIndex + i;
            if (index < this.state.log.length && this.state.log[index].term !==
entries[i].term) {
                // Conflict, delete this and all following
                this.state.log = this.state.log.slice(0, index);
            }
        }
    }

    // Append new entries
    for (let i = 0; i < entries.length; i++) {
        const index = prevLogIndex + i;
        if (index >= this.state.log.length) {
            this.state.log.push(entries[i]);
        }
    }

    // Update commit index
    if (leaderCommit > this.state.commitIndex) {
        this.state.commitIndex = Math.min(leaderCommit, this.state.log.length);
        this.applyCommittedEntries();
    }

    return { term: this.state.currentTerm, success: true };
}

```

```

/**
 * Handle client command
 */
async submitCommand(command: any): Promise<{ committed: boolean; result?: any }> {
  if (this.state.state !== 'leader') {
    return { committed: false }; // Not leader, reject
  }

  // Add entry to log
  const entry: LogEntry = {
    term: this.state.currentTerm,
    index: this.state.log.length,
    command,
  };
  this.state.log.push(entry);

  // Replicate to followers
  await this.replicateToFollowers();

  // Entry is committed when replicated to majority
  const successfulReplications = 1 + this.countSuccessfulReplications(); // +1
  for self
  const majority = Math.floor((this.peers.size + 1) / 2) + 1;

  if (successfulReplications >= majority) {
    this.state.commitIndex = entry.index;
    this.applyCommittedEntries();
    return { committed: true, result: this.stateMachine.get(command.id) };
  }

  return { committed: false };
}

/**
 * Replicate log entries to followers
 */
private async replicateToFollowers(): Promise<void> {
  if (this.state.state !== 'leader') return;

  if (!this.state.leaderNextIndex) {
    // Initialize on first replication
    this.state.leaderNextIndex = new Map();
    this.state.leaderMatchIndex = new Map();
    for (const peerId of this.peers.keys()) {
      this.state.leaderNextIndex.set(peerId, this.state.log.length);
      this.state.leaderMatchIndex!.set(peerId, 0);
    }
  }

  const replications: Promise<boolean>[] = [];

  for (const [peerId, peer] of this.peers) {

```

```

const nextIndex = this.state.leaderNextIndex!.get(peerId)!;
const prevIndex = nextIndex - 1;
const prevTerm = prevIndex > 0 ? this.state.log[prevIndex - 1].term : 0;
const entries = this.state.log.slice(nextIndex);

replications.push(
  peer
    .appendEntries(
      this.state.currentTerm,
      this.nodeId,
      prevIndex,
      prevTerm,
      entries,
      this.state.commitIndex
    )
    .then(result => {
      if (result.success) {
        this.state.leaderMatchIndex!.set(peerId, nextIndex + entries.length -
1);

        this.state.leaderNextIndex!.set(peerId, nextIndex + entries.length);
        return true;
      } else {
        if (result.term > this.state.currentTerm) {
          this.state.currentTerm = result.term;
          this.state.state = 'follower';
        }
        // Decrement nextIndex for this follower
        this.state.leaderNextIndex!.set(peerId, Math.max(1, nextIndex - 1));
        return false;
      }
    })
    .catch(() => false) // Network error
  );
}

await Promise.all(replications);
}

/**
 * Count how many followers successfully replicated
 */
private countSuccessfulReplications(): number {
  if (!this.state.leaderMatchIndex) return 0;
  return this.state.leaderMatchIndex.size; // Simplified, should check match
indices
}

/**
 * Apply committed entries to state machine
 */
private applyCommittedEntries(): void {
  while (this.state.lastApplied < this.state.commitIndex) {

```

```

        this.state.lastApplied++;
        const entry = this.state.log[this.state.lastApplied - 1];
        this.stateMachine.set(entry.command.id, entry.command.value);
    }
}

/**
 * Election timeout - trigger leader election
 */
private onElectionTimeout(): void {
    if (this.state.state !== 'leader') {
        // Convert to candidate and request votes
        this.state.state = 'candidate';
        this.state.currentTerm++;
        this.state.votedFor = this.nodeId;

        this.requestVotesFromPeers();
    }

    this.resetElectionTimer();
}

/**
 * Request votes from all peers
 */
private async requestVotesFromPeers(): Promise<void> {
    const lastEntry = this.getLastLogEntry();
    const votePromises: Promise<{ term: number; voteGranted: boolean }>[] = [];

    for (const [, peer] of this.peers) {
        votePromises.push(
            peer.requestVote(
                this.state.currentTerm,
                this.nodeId,
                lastEntry.index,
                lastEntry.term
            )
        );
    }

    const results = await Promise.all(votePromises);
    let voteCount = 1; // Vote for self

    for (const result of results) {
        if (result.term > this.state.currentTerm) {
            this.state.currentTerm = result.term;
            this.state.state = 'follower';
            return;
        }

        if (result.voteGranted) {
            voteCount++;
        }
    }
}

```

```

    }
}

// Check if won election
const majority = Math.floor((this.peers.size + 1) / 2) + 1;
if (voteCount >= majority && this.state.state === 'candidate') {
    this.becomeLeader();
}
}

/**
 * Become leader
 */
private becomeLeader(): void {
    this.state.state = 'leader';
    this.state.leaderNextIndex = new Map();
    this.state.leaderMatchIndex = new Map();

    // Initialize replication indices
    for (const peerId of this.peers.keys()) {
        this.state.leaderNextIndex!.set(peerId, this.state.log.length);
        this.state.leaderMatchIndex!.set(peerId, 0);
    }

    // Send heartbeats immediately
    this.sendHeartbeats();
    this.resetHeartbeatTimer();
}

/**
 * Send heartbeats to followers
 */
private async sendHeartbeats(): Promise<void> {
    if (this.state.state !== 'leader') return;

    // Send heartbeat/append entries to all followers
    await this.replicateToFollowers();
}

/**
 * Reset election timer
 */
private resetElectionTimer(): void {
    if (this.electionTimeout) {
        clearTimeout(this.electionTimeout);
    }
    this.electionTimeout = setTimeout(() => this.onElectionTimeout(),
this.electionTimeoutMs);
}

/**
 * Reset heartbeat timer

```

```

    */
    private resetHeartbeatTimer(): void {
        if (this.heartbeatInterval) {
            clearInterval(this.heartbeatInterval);
        }
        if (this.state.state === 'leader') {
            this.heartbeatInterval = setInterval(() => this.sendHeartbeats(),
this.heartbeatIntervalMs);
        }
    }

    /**
     * Get last log entry
     */
    private getLastLogEntry(): { index: number; term: number } {
        if (this.state.log.length === 0) {
            return { index: 0, term: 0 };
        }
        const last = this.state.log[this.state.log.length - 1];
        return { index: last.index, term: last.term };
    }

    /**
     * Get current state
     */
    getStatus() {
        return {
            nodeId: this.nodeId,
            state: this.state.state,
            currentTerm: this.state.currentTerm,
            commitIndex: this.state.commitIndex,
            logLength: this.state.log.length,
            lastApplied: this.state.lastApplied,
        };
    }

    /**
     * Get state machine (committed data)
     */
    getStateMachine() {
        return new Map(this.stateMachine);
    }
}

// Example: 5-node Raft cluster
class RaftCluster {
    private nodes: Map<string, RaftNode>;

    constructor(nodeIds: string[]) {
        this.nodes = new Map();

        // Create all nodes

```

```

    for (const id of nodeIds) {
      this.nodes.set(id, new RaftNode(id, this.nodes));
    }
  }

  /**
   * Submit command to cluster (route to leader)
   */
  async submitCommand(command: any): Promise<{ committed: boolean; result?: any }> {
    // Find leader
    const leader = Array.from(this.nodes.values()).find(n => n.getStatus().state ===
'leader');

    if (!leader) {
      return { committed: false }; // No leader
    }

    return leader.submitCommand(command);
  }

  /**
   * Get cluster status
   */
  getStatus() {
    const statuses: Record<string, any> = {};
    for (const [id, node] of this.nodes) {
      statuses[id] = node.getStatus();
    }
    return statuses;
  }

  /**
   * Simulate network partition
   */
  simulatePartition(partition1: string[], partition2: string[]): void {
    // Nodes in partition1 can talk to each other
    // Nodes in partition2 can talk to each other
    // But partition1 and partition2 cannot talk

    // Implementation: Block network calls between partitions
    console.log(`Network partition: [${partition1.join(',')}] vs
[${partition2.join(',')}]`);
  }
}

// Express API for Raft cluster
const cluster = new RaftCluster(['A', 'B', 'C', 'D', 'E']);

app.post('/command', async (req, res) => {
  const result = await cluster.submitCommand(req.body);

  if (result.committed) {

```

```

    res.json({ committed: true, result: result.result });
  } else {
    res.status(503).json({
      committed: false,
      message: 'Cannot process command (no leader or not replicated)',
    });
  }
});

app.get('/status', (req, res) => {
  res.json(cluster.getStatus());
});

```

6. Correct Usage Patterns (Where This Shines)

Pattern 1: Replicated State Machine

```

// All nodes apply same commands in same order
// Results in identical state

await cluster.submitCommand({ operation: 'SET', key: 'user_1', value: { name:
'Alice' } });

// All nodes execute this command:
// stateMachine['user_1'] = { name: 'Alice' }
// Result: Consistent state across all nodes

```

Pattern 2: Log-Based Replication

```

// Command added to log, replicated to majority, then applied
// Safety: Committed entries survive any single node failure

```

Pattern 3: Automatic Failover

```

// Leader fails → Election triggered
// New leader elected from followers
// Clients retry on old leader → Fail → Retry on new leader
// Automatic with no manual intervention

```

7. Failure Modes & Edge Cases

Rapid Leader Changes

Problem: Leader fails, new leader elected, but old leader comes back.

Mitigation: Term numbers ensure old leader deposed (term too low).

Log Divergence During Partition

Problem: Minority partition has uncommitted entries not replicated to majority.

Mitigation: New leader elected from majority, discards old leader's uncommitted entries.

Network Delays vs Failures

Problem: Slow network looks like node failure.

Mitigation: Conservative timeouts, multiple election rounds.

8. Performance Characteristics & Tradeoffs

Consistency

Guarantee: Strong consistency (linearizability)

Cost: Must replicate to majority before committing (higher latency)

Availability

During partition: Minority partition unavailable (reduced availability)

After partition heals: Automatic recovery, consistent state

Latency

Write latency: RTT to majority + disk fsync (must replicate + persist)

Read latency: Local read (any node can serve reads from committed entries)

9. Foot-Guns & Common Mistakes (DO NOT SKIP)

Mistake 1: Not Checking Election Term in Append Entries

Fix: Reject messages from old terms.

Mistake 2: Forgetting to Persist Vote

Fix: Write votedFor to disk before responding to vote request.

Mistake 3: Committing Entries from Old Terms

Fix: Only commit entries replicated in current term.

Mistake 4: Not Handling Log Conflicts

Fix: Detect and delete conflicting entries from followers.

Mistake 5: No Snapshot/Compaction Mechanism

Fix: Implement log snapshotting to prevent infinite log growth.

10. When NOT to Use This (Anti-Patterns)

Anti-Pattern 1: Consensus for Every Decision

Raft is for state replication. Don't use for ephemeral decisions.

Anti-Pattern 2: Byzantine Faults

Raft doesn't handle malicious nodes.

Anti-Pattern 3: Cross-Datacenter Raft

High latency across data centers hurts Raft performance.

11. Related Concepts (With Contrast)

Paxos

Difference: Both solve consensus. Raft is simpler to understand and implement.

Quorum-Based Replication

Difference: Quorum provides safety but not full consensus (doesn't handle log divergence).

Two-Phase Commit

Difference: 2PC coordinates transactions. Raft coordinates state machine state.

12. Production Readiness Checklist

Core Algorithm

- ☐ Election implemented (term-based)
- ☐ Log replication implemented
- ☐ Commit point tracking
- ☐ Safety checks (vote once per term, log matching)

Persistence

- ☐ Current term persisted
- ☐ Voted for persisted
- ☐ Log persisted
- ☐ State machine persisted

Failure Handling

- ☐ Follower handles leader failure (election timeout)
- ☐ Leader handles follower failure (retry, next index decrement)
- ☐ Handle network partitions (minority partition blocks)

Monitoring

- ☐ Track leadership changes
- ☐ Track replication lag
- ☐ Monitor log size
- ☐ Alert on no leader

Testing

- ☐ Unit tests for all RPCs

- ☐ Integration tests with cluster
- ☐ Network partition tests
- ☐ Leader failure + recovery tests