Leader Election

1. The Real Problem This Exists to Solve

Distributed systems need one authoritative decision-maker (leader) to coordinate work, prevent conflicts, and make progress. Leader election is the process of choosing that leader automatically when the current leader fails, ensuring the system always has exactly one active leader and can recover from failures without manual intervention.

Real production scenario:

- Database replication cluster elects primary
- Normal operation: Primary A handles all writes
- Primary A fails (hardware crash)
- **Without leader election:**
  - Manually detect failure (5 minutes)
  - Manually choose new primary (10 minutes)
  - Manually update config in all clients (15 minutes)
  - **Total downtime: 30 minutes**
  - Clients manually failover (need human action)
  - **Customer impact: 30 minutes of service unavailable**
- **With automatic leader election:**
  - Followers detect leader unreachable (1-2 seconds)
  - Followers hold election (300-500ms)
  - New primary elected and starts accepting writes
  - Clients auto-discover new primary (via health checks)
  - **Total downtime: 1-2 seconds**
  - Automatic recovery, no human action needed
  - **Customer impact: Brief delay, then continues**

The fundamental problem: Choosing a leader when current leader is unknown to be alive or dead is difficult in an unreliable network. Multiple nodes may simultaneously believe they should be leader, causing conflicts and data corruption.

Without leader election:

- Manual intervention required for every failure
- Long outages (tens of minutes)
- Human error possible (choose wrong node)
- Cannot recover from cascading failures
- Not scalable for frequent failures

With leader election:

- Automatic failure detection and recovery
- Seconds to minutes of downtime (not hours)
- No human intervention needed
- Cascading failures handled automatically
- Scales to any cluster size

2. The Naive / Incorrect Approaches (IMPORTANT)

❌ Incorrect Approach #1: Ring/Bully Algorithm With No Timeout

```
// Incorrect: Bully algorithm without proper timeout handling
class BullyElection {
  private leaderId: string | null = null;
  private nodeIds = ['A', 'B', 'C', 'D', 'E'];

  async electLeader(): Promise<string> {
    // Start election: "Anyone with higher ID, speak up"
    const candidates: string[] = [];

    for (const nodeId of this.nodeIds) {
      if (nodeId > (this.leaderId || '')) {
        try {
          const response = await this.ping(nodeId);
          if (response) candidates.push(nodeId);
        } catch {
          // Node unreachable, skip
        }
      }
    }

    if (candidates.length === 0) {
      // No higher node responded
      this.leaderId = this.nodeIds[this.nodeIds.length - 1];
    } else {
      // Wait for highest to claim leadership
      // But no timeout if it never does!
    }

    return this.leaderId!;
  }

  private async ping(nodeId: string): Promise<boolean> {
    // Might hang if network broken
    return await this.rpc(nodeId, 'ping');
  }
}

// Problem: If higher node alive but slow, election never completes
// Or if higher node becomes leader but then dies, no re-election
```

Why it seems reasonable:

- Simple algorithm (well-known)
- Intuitive (higher ID always wins)

How it breaks:

- No timeout on wait → infinite hanging
- If elected leader dies, no re-election triggered
- If network slow, election never completes

- Multiple elections happening simultaneously → conflicts

Production symptoms:

- System hangs waiting for election result
- Leader dies but system doesn't notice (no periodic checks)
- Multiple leaders elected (from different election rounds)

❌ Incorrect Approach #2: Random Backoff Without Coordination

```
// Incorrect: Random backoff for election, no coordination
class RandomBackoffElection {
  private leaderId: string | null = null;

  async electLeader(): Promise<string> {
    // If no leader detected, each node independently
    // waits a random time before claiming leadership
    const backoff = Math.random() * 1000;  // 0-1 second
    await this.sleep(backoff);

    // Claim leadership
    this.leaderId = this.nodeId;
    return this.leaderId;
  }
}

// Problem: Two nodes with similar backoff times both claim leadership
// No mechanism to ensure only one claims
// And both might successfully claim leadership
```

Why it seems reasonable:

- Avoids synchronized election (staggered backoff)
- Simple to implement

How it breaks:

- Two nodes with similar random delays both become leader
- No coordination between nodes
- Each node independently decides (no quorum)
- Multiple leaders possible

Production symptoms:

- Two leaders claim authority simultaneously
- Data divergence (both accept writes)
- No automatic conflict resolution

❌ Incorrect Approach #3: IP Address-Based Leader Selection

```
// Incorrect: Always choose highest/lowest IP as leader
class StaticLeaderSelection {
  private nodeIps = ['192.168.1.1', '192.168.1.2', '192.168.1.3'];
```

```
  selectLeader(): string {
    // Always pick highest IP
    return this.nodeIps.sort().reverse()[0];  // Deterministic
  }
}

// Problem: Elected leader might be down
// No mechanism to detect failure and elect new leader
```

Why it seems reasonable:

- Deterministic (same leader always chosen)
- Simple
- No randomness

How it breaks:

- If elected leader is down, no re-election
- System broken until leader comes back or human intervention
- No failure detection built in

Production symptoms:

- Hardcoded leader is down
- System completely broken (no failover)
- Requires manual config change to select new leader

❌ Incorrect Approach #4: Heartbeat Without Quorum Check

```
// Incorrect: Heartbeat seen from any node = leader is alive
class HeartbeatOnlyElection {
  private leaderId: string | null = 'A';
  private lastHeartbeat = Date.now();

  async checkLeaderHealth(): Promise<void> {
    const timeSinceHeartbeat = Date.now() - this.lastHeartbeat;

    if (timeSinceHeartbeat > 5000) {  // 5 second timeout
      // Leader not responding
      // But don't check with other nodes!
      this.leaderId = null;  // Mark as down
    }
  }

  async receiveHeartbeat(leaderId: string): Promise<void> {
    // Any heartbeat = leader is alive
    this.leaderId = leaderId;
    this.lastHeartbeat = Date.now();
  }
}

// Problem: Network partition
// Partition 1 (containing leader A, nodes B, C):
```

```
//   All see leader A, no re-election
// Partition 2 (containing nodes D, E):
//   Don't see A's heartbeat, elect new leader E
// Result: 2 leaders (A and E)
```

Why it seems reasonable:

- Direct detection (if see heartbeat, leader is alive)
- Simple

How it breaks:

- During partition: Minority partition doesn't see leader heartbeat
- Minority independently elects new leader
- Network heals: Two leaders exist
- No split-brain prevention (quorum check)

Production symptoms:

- Network partition triggers two elections
- Two leaders in different partitions
- Data divergence when partition heals

3. Correct Mental Model (How It Actually Works)

Leader election is a distributed algorithm that ensures exactly one node claims leadership in each term/epoch, with majority quorum agreement, and automatically triggers re-election when leader fails or partitions occur.

The Election Requirements

```
Valid election must satisfy:

1. Exactly One Leader Per Term
   - No two nodes both leaders in same term
   - Enforced by: Only one can get quorum votes

2. Quorum Agreement
   - Leader must have majority quorum votes
   - No minority can independently elect leader

3. Failure Detection
   - Detect when leader is unreachable
   - Trigger re-election automatically

4. Safety
   - Newly elected leader has latest data
   - No data loss during failover
```

The Election Phases

```
Phase 1: Failure Detection
  Follower: "Leader not responding for 1 second"
  Action: Enter election mode
```

```
Phase 2: Candidate Campaign
  Candidate: "Vote for me, I'm candidate for term N"
  Sends request to all other nodes

Phase 3: Vote Collection
  Other nodes: "I'll vote for you if you're qualified"
  Candidate: Count votes

Phase 4: Victory or Defeat
  If majority votes: Become leader
  If another wins: Accept new leader, back to follower
  If split: Restart election with new term
```

The Qualification Rules

```
Follower only votes once per term
  → Prevents same term election from splitting votes

Candidate must have up-to-date log
  → Ensures new leader has latest data

Only higher term can replace current leader
  → Ensures old leader doesn't come back

Leader sends heartbeats to maintain authority
  → Resets election timer on followers
```

The Cluster Sizes

```
Cluster Size  Quorum  Can Tolerate Failures
2             2       0 (any failure = no quorum)
3             2       1 node failure
4             3       1 node failure
5             3       2 node failures
7             4       3 node failures

Key: Odd number of nodes is optimal
(n nodes needs (n/2)+1 for quorum)
```

4. Correct Design & Algorithm

Raft Leader Election Algorithm

```
Follower state:
  - Election timeout = 150-300ms randomized
  - If no heartbeat from leader for timeout: Go to candidate

Candidate state:
  - Increment term
```

```
  - Vote for self
  - Request vote from all other nodes
  - Wait for responses:
    - If majority votes: Become leader
    - If see higher term: Back to follower
    - If election timeout: Start new election (higher term)

Leader state:
  - Send heartbeats periodically (50ms)
  - Heartbeat resets followers' election timer
  - If see higher term: Back to follower
```

Safety Rules

```
Vote only once per term
  Each node tracks: "Which term did I vote in, who did I vote for"
  On new vote request: If term higher, can vote again. If term same, reject if already
voted.

Check candidate qualification
  Candidate must have term >= current term
  Candidate must have up-to-date log (at least as recent as voter)

Only one leader per term
  Result of quorum voting + vote-once-per-term
```

5. Full Production-Grade Implementation

```typescript
interface ElectionState {
  currentTerm: number;
  votedFor: string | null;
  state: 'follower' | 'candidate' | 'leader';
  leaderId: string | null;
  lastHeartbeatTime: number;
  lastLogTerm: number;
  lastLogIndex: number;
}

class LeaderElection {
  private nodeId: string;
  private peers: string[];
  private state: ElectionState;
  private readonly electionTimeoutMin = 150;
  private readonly electionTimeoutMax = 300;
  private readonly heartbeatInterval = 50;
  private electionTimer: NodeJS.Timeout | null = null;
  private heartbeatTimer: NodeJS.Timeout | null = null;
  private electionTimeout: number;

  constructor(nodeId: string, peers: string[]) {
```

```
    this.nodeId = nodeId;
    this.peers = peers;
    this.electionTimeout = this.randomElectionTimeout();
    this.state = {
      currentTerm: 0,
      votedFor: null,
      state: 'follower',
      leaderId: null,
      lastHeartbeatTime: Date.now(),
      lastLogTerm: 0,
      lastLogIndex: 0,
    };

    this.startElectionTimer();
  }

  /**
   * Request vote RPC (candidate requests vote from follower)
   */
  async requestVote(
    term: number,
    candidateId: string,
    lastLogTerm: number,
    lastLogIndex: number
  ): Promise<{ term: number; voteGranted: boolean }> {
    // Rule 1: If request term < current term, reject
    if (term < this.state.currentTerm) {
      return { term: this.state.currentTerm, voteGranted: false };
    }

    // Rule 2: If request term > current term, update and reset vote
    if (term > this.state.currentTerm) {
      this.state.currentTerm = term;
      this.state.votedFor = null;
      if (this.state.state !== 'follower') {
        this.becomeFollower();
      }
    }

    // Rule 3: If already voted in this term for different candidate, reject
    if (this.state.votedFor !== null && this.state.votedFor !== candidateId) {
      return { term: this.state.currentTerm, voteGranted: false };
    }

    // Rule 4: Check if candidate log is up-to-date
    // Candidate must have term >= my last log term
    // Or (term == my term AND index >= my last index)
    if (lastLogTerm < this.state.lastLogTerm) {
      return { term: this.state.currentTerm, voteGranted: false };
    }
    if (lastLogTerm === this.state.lastLogTerm && lastLogIndex <
this.state.lastLogIndex) {
```

```
    return { term: this.state.currentTerm, voteGranted: false };
  }

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

/**
 * Append entries RPC (leader sends heartbeat)
 */
async appendEntries(
  term: number,
  leaderId: string
): Promise<{ term: number; success: boolean }> {
  // Update term if needed
  if (term > this.state.currentTerm) {
    this.state.currentTerm = term;
    this.state.votedFor = null;
    if (this.state.state !== 'follower') {
      this.becomeFollower();
    }
  }

  // If request from old term, reject
  if (term < this.state.currentTerm) {
    return { term: this.state.currentTerm, success: false };
  }

  // Accept heartbeat from leader
  this.state.leaderId = leaderId;
  this.state.lastHeartbeatTime = Date.now();

  if (this.state.state !== 'follower') {
    this.becomeFollower();
  }

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

/**
 * Handle election timeout - start new election
 */
private onElectionTimeout(): void {
  if (this.state.state !== 'leader') {
    this.becomeCandidate();
  }
}

/**
 * Become candidate and request votes
```

```
 */
private async becomeCandidate(): Promise<void> {
  this.state.state = 'candidate';
  this.state.currentTerm++;
  this.state.votedFor = this.nodeId;  // Vote for self
  this.state.leaderId = null;

  console.log(
    `[${this.nodeId}] Became candidate for term ${this.state.currentTerm}`
  );

  // Request votes from all peers
  await this.requestVotesFromPeers();

  // Restart election timer in case this election fails
  this.resetElectionTimer();
}

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

  // Request vote from each peer
  for (const peerId of this.peers) {
    votePromises.push(
      this.sendRequestVote(
        peerId,
        this.state.currentTerm,
        this.nodeId,
        this.state.lastLogTerm,
        this.state.lastLogIndex
      ).catch(() => ({ term: this.state.currentTerm, voteGranted: false }))
    );
  }

  const results = await Promise.all(votePromises);

  let voteCount = 1;  // Vote for self

  for (const result of results) {
    // If see higher term, step down
    if (result.term > this.state.currentTerm) {
      this.state.currentTerm = result.term;
      this.becomeFollower();
      return;
    }

    // Count votes
    if (result.voteGranted) {
      voteCount++;
```

```
      }
    }

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

  /**
   * Become leader
   */
  private becomeLeader(): void {
    this.state.state = 'leader';
    this.state.leaderId = this.nodeId;

    console.log(`[${this.nodeId}] Became leader for term
${this.state.currentTerm}`);

    // Send heartbeats immediately
    this.sendHeartbeats();

    // Start heartbeat timer
    if (this.heartbeatTimer) {
      clearInterval(this.heartbeatTimer);
    }
    this.heartbeatTimer = setInterval(() => this.sendHeartbeats(),
this.heartbeatInterval);
  }

  /**
   * Become follower
   */
  private becomeFollower(): void {
    if (this.state.state === 'leader') {
      console.log(`[${this.nodeId}] Stepped down from leader role`);

      if (this.heartbeatTimer) {
        clearInterval(this.heartbeatTimer);
        this.heartbeatTimer = null;
      }
    }

    this.state.state = 'follower';
    this.state.leaderId = null;
    this.resetElectionTimer();
  }

  /**
   * Send heartbeats to all followers
   */
```

```
private async sendHeartbeats(): Promise<void> {
  if (this.state.state !== 'leader') return;

  const promises: Promise<void>[] = [];

  for (const peerId of this.peers) {
    promises.push(
      this.sendAppendEntries(peerId, this.state.currentTerm, this.nodeId)
        .then(result => {
          if (result.term > this.state.currentTerm) {
            this.state.currentTerm = result.term;
            this.becomeFollower();
          }
        })
        .catch(() => {})  // Ignore network errors
    );
  }

  await Promise.all(promises);
}

/**
 * RPC: Request vote from peer
 */
private async sendRequestVote(
  peerId: string,
  term: number,
  candidateId: string,
  lastLogTerm: number,
  lastLogIndex: number
): Promise<{ term: number; voteGranted: boolean }> {
  // Simulated RPC (would be actual network call)
  return { term, voteGranted: Math.random() > 0.5 };
}

/**
 * RPC: Send append entries (heartbeat)
 */
private async sendAppendEntries(
  peerId: string,
  term: number,
  leaderId: string
): Promise<{ term: number; success: boolean }> {
  // Simulated RPC
  return { term, success: true };
}

/**
 * Reset election timer
 */
private resetElectionTimer(): void {
  if (this.electionTimer) {
```

```
      clearTimeout(this.electionTimer);
    }

    this.electionTimeout = this.randomElectionTimeout();
    this.electionTimer = setTimeout(() => this.onElectionTimeout(),
this.electionTimeout);
  }

  /**
   * Random election timeout (150-300ms)
   */
  private randomElectionTimeout(): number {
    return (
      this.electionTimeoutMin +
      Math.random() * (this.electionTimeoutMax - this.electionTimeoutMin)
    );
  }

  /**
   * Start election timer
   */
  private startElectionTimer(): void {
    this.resetElectionTimer();
  }

  /**
   * Get election state
   */
  getState() {
    return {
      nodeId: this.nodeId,
      state: this.state.state,
      currentTerm: this.state.currentTerm,
      leaderId: this.state.leaderId,
      votedFor: this.state.votedFor,
      lastHeartbeat: Date.now() - this.state.lastHeartbeatTime,
    };
  }
}

// Cluster of multiple nodes
class ElectionCluster {
  private nodes: Map<string, LeaderElection>;
  private clusterMembers: string[];

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

    // Create all nodes
    for (const nodeId of nodeIds) {
      const peers = nodeIds.filter(id => id !== nodeId);
```

```
      this.nodes.set(nodeId, new LeaderElection(nodeId, peers));
    }
  }

  /**
   * Get cluster state
   */
  getClusterState() {
    const states: Record<string, any> = {};
    for (const [nodeId, node] of this.nodes) {
      states[nodeId] = node.getState();
    }
    return states;
  }

  /**
   * Get current leader
   */
  getLeader(): string | null {
    for (const [nodeId, node] of this.nodes) {
      const state = node.getState();
      if (state.state === 'leader') {
        return nodeId;
      }
    }
    return null;
  }

  /**
   * Simulate leader failure
   */
  killLeader(): void {
    const leader = this.getLeader();
    if (leader) {
      console.log(`[SIM] Killing leader ${leader}`);
      // In real implementation, would stop heartbeats and become unreachable
    }
  }

  /**
   * Simulate network partition
   */
  simulatePartition(partition1: string[], partition2: string[]): void {
    console.log(`[SIM] Partition: [${partition1.join(',')}] vs
[${partition2.join(',')}]`);
    // Nodes in partition1 can't reach partition2
  }
}

// Express API
const cluster = new ElectionCluster(['A', 'B', 'C', 'D', 'E']);
```

```
app.get('/election/status', (req, res) => {
  res.json({
    clusterState: cluster.getClusterState(),
    currentLeader: cluster.getLeader(),
  });
});

app.post('/simulation/kill-leader', (req, res) => {
  cluster.killLeader();
  res.json({ message: 'Killed leader, new election in progress' });
});

app.post('/simulation/partition', (req, res) => {
  const { partition1, partition2 } = req.body;
  cluster.simulatePartition(partition1, partition2);
  res.json({ message: 'Partition simulated' });
});
```

6. Correct Usage Patterns (Where This Shines)

Pattern 1: Automatic Failover

```
// When leader dies, election triggered automatically
// Minority partition cannot elect new leader (no quorum)
// Majority partition elects new leader (has quorum)
// Clients retry on old leader, get redirected to new leader
```

Pattern 2: Cascading Failures

```
// Primary fails → Secondary elected
// Secondary fails → Tertiary elected
// System always has exactly one leader
// No manual intervention needed
```

Pattern 3: Partition Recovery

```
// Partition heals
// Two leaders exist (one from each partition)
// Higher term leader wins (old leader steps down)
// Minority catches up from majority
```

7. Failure Modes & Edge Cases

Split Votes

Problem: Candidates split votes evenly, no one gets majority.

Mitigation: Random election timeout ensures re-election with different timeout.

Leader Loses Quorum

Problem: Leader becomes partitioned (in minority).

Mitigation: Leader checks if has quorum before accepting writes.

Asymmetric Partition

Problem: One-way network failure (A→B works, B→A fails).

Mitigation: Require bidirectional communication (heartbeat response).

8. Performance Characteristics & Tradeoffs

Detection Time

Fast timeout: Quick failure detection (150ms) but more false positives

Slow timeout: Fewer false positives but slower failure detection (>1s)

Election Time

Depends on: Timeout + RPC latency + vote collection

Typical: 300-500ms for full failover

Availability During Election

Minority partition: Unavailable (no leader possible)

Majority partition: Available (new leader will be elected)

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

Mistake 1: Synchronized Election Timeout

Fix: Randomize timeout (150-300ms) to avoid tied elections.

Mistake 2: Forgetting to Update Term on Higher Request

Fix: Always check term, step down if see higher term.

Mistake 3: No Vote Persistence

Fix: Persist votedFor to disk (survive crashes).

Mistake 4: Leader Doesn't Check Quorum

Fix: Leader must verify quorum before accepting writes.

Mistake 5: No Heartbeat Timeout

Fix: Implement heartbeat timeout (follower detects leader failure).

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

Anti-Pattern 1: Leader Election for Every Decision

Use for consistent leader role, not for ephemeral decisions.

Anti-Pattern 2: Too-Long Election Timeout

Don't make timeout so long that failures aren't detected quickly.

Anti-Pattern 3: Two-Node Cluster

Two nodes cannot form quorum (need 2 for majority). Use at least 3.

11. Related Concepts (With Contrast)

Consensus (Raft)

Difference: Leader election chooses leader. Consensus replicates log.

Quorum Voting

Difference: Quorum voting chooses among options. Leader election chooses exactly one leader per term.

Health Checks

Difference: Health checks detect failures. Election chooses replacement.

12. Production Readiness Checklist

Core Algorithm

- [ ] Randomized election timeout implemented
- [ ] Vote-once-per-term enforced
- [ ] Quorum voting for leader
- [ ] Term numbers tracked

Failure Detection

- [ ] Election timeout triggers re-election
- [ ] Leader sends heartbeats periodically
- [ ] Heartbeat resets election timer
- [ ] No elected leader = followers wait for new election

Safety

- [ ] Only quorum can elect leader
- [ ] Minority partition cannot elect
- [ ] Old leaders step down on higher term
- [ ] Exactly one leader per term

Persistence

- [ ] Current term persisted
- [ ] Voted for persisted

- [ ] Recover state after crash

Monitoring

- [ ] Track leadership changes
- [ ] Alert on frequent elections
- [ ] Monitor election duration
- [ ] Metrics on split votes