Split Brain & Network Partitions

1. The Real Problem This Exists to Solve

When a network fails between components of a distributed system, each partition may believe itself to be the healthy part of the system and continue operating independently. This causes data divergence, conflicting writes, and inconsistent state. Split brain prevention ensures that the system chooses a single authority or stops accepting writes entirely.

Real production scenario:

- Database replication cluster (primary + 2 replicas across data centers)
- Primary in US-East, Replicas in US-West and EU
- Network partition: US-East becomes isolated from US-West and EU
- **Without split brain prevention:**
  - US-East primary: "US-West and EU disconnected, I'm still primary"
  - US-West detects US-East disconnect: "Primary is down, I'm promoting myself to primary"
  - EU also detects disconnect: "Both others are down, I'm promoting to primary"
  - Now 3 database primaries: writing independently in all 3 data centers
  - User writes to US-East primary: order placed
  - User writes to US-West primary: different order placed
  - Network heals: conflicting writes detected
  - Data corruption: Both orders exist, billing charged twice
  - **Revenue impact: Reconciliation costs, customer refunds, regulatory issues**
- **With split brain prevention:**
  - US-East primary: "I detect partition, but I'm only 1 of 3 nodes"
  - US-West and EU: Both see US-East gone, together they're 2/3 (quorum)
  - US-West and EU form new primary: "We have quorum"
  - US-East: "I lost quorum, cannot accept writes"
  - Writes rejected in US-East with "not primary" error
  - US-West/EU primary continues accepting writes (client failover required)
  - Network heals: Automatic reconciliation from quorum primary
  - **Result: No data corruption, clean failover, clients retry successfully**

The fundamental problem: In a distributed system, network partition means each partition lacks complete information. If each partition independently assumes "others are down, I'm healthy," conflicting decisions occur leading to data corruption.

Without split brain prevention:

- Multiple independent primaries possible
- Writes accepted in multiple locations simultaneously
- Data diverges permanently (conflicting writes)
- Data corruption when network heals
- No automatic resolution possible

With split brain prevention:

- Only one partition can be "active"
- Minority partition cannot write
- Majority partition continues normally
- No conflicting writes possible

- Automatic healing when network recovers

2. The Naive / Incorrect Approaches (IMPORTANT)

❌ Incorrect Approach #1: Timeout-Based Failover Without Quorum

```
// Incorrect: Each node independently decides primary is down
class ReplicaNode {
  private primaryHealthCheckInterval = 5000;  // 5 second timeout
  private primaryLastSeen = Date.now();

  constructor(private isPrimary = false) {
    setInterval(() => this.checkPrimaryHealth(), this.primaryHealthCheckInterval);
  }

  private checkPrimaryHealth(): void {
    try {
      pingPrimary();
      this.primaryLastSeen = Date.now();
    } catch {
      const timeSinceSeen = Date.now() - this.primaryLastSeen;

      if (timeSinceSeen > this.primaryHealthCheckInterval) {
        // Primary not responding, promote self
        this.promoteToPrimary();
      }
    }
  }

  private promoteToPrimary(): void {
    console.log('Primary timeout, promoting to primary');
    this.isPrimary = true;
    // Start accepting writes
  }

  async write(data: any): Promise<void> {
    if (this.isPrimary) {
      // Write to local database
      await db.insert(data);
    } else {
      throw new Error('Not primary');
    }
  }
}
```

Why it seems reasonable:

- Simple to implement
- Detects primary failures automatically
- All replicas can failover independently
- No coordination overhead

How it breaks:

- During network partition, ALL replicas independently see primary unreachable
- All replicas independently promote themselves to primary
- Within 5 seconds: 3 primaries in 3 locations writing independently
- No coordination between replicas
- Example sequence:
    1. Network partition at T=0
    2. T=5: Replica A times out, becomes primary, starts writing
    3. T=5: Replica B times out, becomes primary, starts writing
    4. T=5: Replica C times out, becomes primary, starts writing
    5. Writes diverge: A, B, C accept conflicting writes
    6. T=30: Network heals, 3 conflicting databases
- No way to resolve which was "correct"

Production symptoms:

- Multiple primaries detected in different data centers
- Data diverges: Users in different regions see different data
- Same user ID exists with different values in different replicas
- Network heals but database still corrupted
- Requires manual intervention to select "correct" copy
- Data loss or data duplication inevitable
- Hours/days to reconcile conflicts

❌ Incorrect Approach #2: Quorum Check Without Enforcement

```
// Incorrect: Check for quorum but still write in minority
class Node {
  private nodeIds = ['A', 'B', 'C'];
  private aliveNodes = new Set(['A', 'B', 'C']);

  async write(data: any): Promise<void> {
    // Check if we have quorum (2/3)
    const quorum = Math.floor(this.nodeIds.length / 2) + 1;

    if (this.aliveNodes.size >= quorum) {
      console.log('Have quorum, writing');
      await this.doWrite(data);
    } else {
      console.log('No quorum, but writing anyway');
      // BUG: Still write even without quorum!
      await this.doWrite(data);
    }
  }

  private async doWrite(data: any): Promise<void> {
    await db.insert(data);
    // Replicate to other nodes
    for (const nodeId of this.nodeIds) {
      if (this.aliveNodes.has(nodeId)) {
        await replicateTo(nodeId, data);
```

```
      }
    }
  }
}
```

Why it seems reasonable:

- Checks for quorum (aware of concept)
- Logs quorum status
- Attempts to validate before writing

How it breaks:

- Checks for quorum but ignores result (still writes anyway)
- "Have quorum" logic just informational, not enforced
- During partition: All nodes log "no quorum" but write anyway
- Same outcome as Approach #1 (multiple primaries)
- False sense of safety from quorum check

Production symptoms:

- Logs show "No quorum" but data still written
- Users confused: quorum check passed but data corrupted
- More insidious than #1 (appears to have safeguard)

❌ Incorrect Approach #3: Quorum Check Without Split Detection

```
// Incorrect: Only check if we have quorum, not if split
class Node {
  private nodeIds = ['A', 'B', 'C'];
  private aliveNodes = new Set(['A', 'B', 'C']);

  async write(data: any): Promise<void> {
    // Count alive nodes including self
    const aliveCount = this.aliveNodes.size + 1;
    const quorum = Math.floor(this.nodeIds.length / 2) + 1;

    if (aliveCount >= quorum) {
      await this.doWrite(data);
    } else {
      throw new Error('Cannot write, no quorum');
    }
  }
}

// Split brain scenario
// Partition 1 (Node A + B): Can talk to each other
//   Count: 2 alive (A, B) + 1 self = 3 ≥ 2 quorum ✓ Can write
// Partition 2 (Node C): Alone
//   Count: 0 alive + 1 self = 1 < 2 quorum ✗ Cannot write
//
```

```
// Looks good! But problem: Node C could be "majority partition" if
// network partitions differently (e.g., A and B lose all networks)
```

Why it seems reasonable:

- Enforces quorum check
- Prevents minority from writing
- Simple logic

How it breaks:

- Assumes network partition is between nodes we know about
- Doesn't distinguish between:
  - "Partition 1 with 2/3 nodes (quorum)"
  - "Partition 1 with 2/3 nodes but isolated from others (might be minority)"
- Example scenario:
  - Node A becomes partition (loses ability to reach B and C)
  - Node A checks: "B and C unreachable, count = 1, no quorum, can't write" ✓
  - Meanwhile Node B and C are also partitioned (A unreachable)
  - Node B checks: "A and C connected, count = 2, quorum, write" ✓
  - Both A AND B become primaries (each assumes the other is down)
- Failed to detect that ALL nodes might be partitioned

Production symptoms:

- Quorum check prevents some failures (good)
- But not all (two disjoint 2-node partitions both have "quorum")
- Still possible for multiple primaries (if partition isn't clean)

❌ Incorrect Approach #4: No Fencing of Old Primary

```
// Incorrect: New primary elected but old primary still writes
class Replica {
  private isPrimary = false;
  private generation = 0;

  async promoteToP primary(): Promise<void> {
    this.isPrimary = true;
    this.generation++;
    console.log(`Promoted to primary, generation ${this.generation}`);
  }

  async write(data: any): Promise<void> {
    if (this.isPrimary) {
      await db.insert(data);
      // Broadcast to replicas with generation number
      await replicateToAll(data, this.generation);
    }
  }
}

// Split brain scenario (no fence mechanism)
```

```
// Original primary P (in isolated partition)
//   - Still thinks it's primary
//   - Writes with generation 1
//
// New primary N (in majority partition)
//   - Promoted to primary
//   - Writes with generation 2
//
// Both P and N writing simultaneously!
// No mechanism prevents P from writing
```

Why it seems reasonable:

- Includes generation numbers
- Newer generation should win
- Seems logical

How it breaks:

- Generation number only helps with reconciliation
- Doesn't prevent old primary from writing
- Both old and new primary accept writes for extended period
- Reconciliation required: "Which writes happened?"
- Generation alone doesn't guarantee consistency

Production symptoms:

- Old primary continues writing even after new primary elected
- Conflicting writes from both primaries for minutes/hours
- New primary must somehow detect and reject old primary writes
- Reconciliation complex and error-prone

3. Correct Mental Model (How It Actually Works)

Split brain is prevented by ensuring only one partition can be active at any time, enforced through quorum rules and fencing.

The Partition Reality

```
Network partition means: Network is partitioned into separate graphs

3-node system, clean partition:
  Partition A: {Node 1, Node 2}
  Partition B: {Node 3}

  Partition A: 2/3 nodes = quorum ✓ Can be primary
  Partition B: 1/3 nodes = minority ✗ Cannot be primary

3-node system, complex partition (cascading):
  Initial: All nodes connected
  T=0: Node A loses network (both outbound and inbound)
  Partition A: {Node A alone}
  Partition B: {Node B, Node C connected to each other}
```

```
  Partition A: 1/3 = no quorum
  Partition B: 2/3 = quorum
```

The Quorum Guarantee

```
Key property: At most one partition can have quorum

For N nodes, quorum = (N/2) + 1

3 nodes: quorum = 2
  Partition sizes: 3-0 (one has quorum)
  Partition sizes: 2-1 (one has quorum)
  IMPOSSIBLE: 1-1-1 or similar (would require both to have quorum)

5 nodes: quorum = 3
  Partition sizes: 5-0 (one has quorum)
  Partition sizes: 4-1 (one has quorum)
  Partition sizes: 3-2 (one has quorum)
  IMPOSSIBLE: 2-2-1 (neither 2-node partition has quorum)
```

The Fencing Mechanism

```
Majority partition needs to prevent minority from writing:

Method 1: Timeout
  - Minority partition waits timeout
  - After timeout, can write (expects partition healed)
  - If not healed, still writes (eventually)

Method 2: Explicit Fencing (writes blocked)
  - Majority partition sends "you are not primary" to minority
  - Minority nodes reject further writes
  - Even if minority partition can talk to each other

Method 3: Quorum Revocation
  - Majority partition revokes minority nodes' write privileges
  - Minority cannot write even if partition heals

Method 4: Generation/Epoch Numbers
  - Each leader has generation number
  - Replicas only accept writes from current generation
  - New leader increments generation
  - Old leader's writes rejected (generation too low)
```

The Healing Process

```
During partition:
  Primary partition: Accepts writes with generation N
  Minority partition: Blocked from writing
```

```
After partition heals:
  Minority catches up from primary
  Apply all writes with generation N
  Both partitions consistent again
  No data loss, no data corruption
```

4. Correct Design & Algorithm

Strategy 1: Quorum-Based Authority

```
N nodes, quorum Q = (N/2) + 1

On write request:
  Check: Do I have quorum of reachable nodes?
  Yes → Accept write
  No → Reject write

On network heal:
  Minority partition discovers primary partition
  Accept all writes from primary partition
  Bring itself up to date
```

Strategy 2: Generation-Based Fencing

```
Each leader has generation number (incremented on each failover)

On write request:
  Leader: Write with generation G
  Replica: Accept only if G >= current_generation

When old leader partitioned:
  New leader: generation G+1
  Old leader: generation G (stale)
  Old leader writes: Rejected (G < G+1)
```

Strategy 3: Lease-Based Fencing

```
Primary must renew lease before writing:

On election:
  Primary gets exclusive lease (e.g., 10 seconds)

On write:
  If lease valid: Accept write
  If lease expired: Reject write

Minority partition:
  Cannot renew lease (majority partition would detect and not grant)
```

```
  Lease expires
  Cannot write
```

Strategy 4: Quorum Reads and Writes

```
On write:
  Write to majority quorum of replicas
  Return only when quorum acknowledges

On read:
  Read from majority quorum of replicas
  Return only if all replicas agree (prevents stale reads)
```

5. Full Production-Grade Implementation

```typescript
interface Node {
  id: string;
  healthy: boolean;
}

interface LeaderState {
  isLeader: boolean;
  generation: number;
  leaseExpiry: number;
  quorumSize: number;
}

class DistributedSystem {
  private nodes: Map<string, Node>;
  private state: LeaderState;
  private heartbeatInterval = 1000;  // 1 second
  private leaseTimeout = 5000;       // 5 second lease
  private replicaData: Map<string, any> = new Map();
  private generation = 0;

  constructor(nodeIds: string[]) {
    this.nodes = new Map(nodeIds.map(id => [id, { id, healthy: true }]));
    this.state = {
      isLeader: false,
      generation: 0,
      leaseExpiry: 0,
      quorumSize: Math.floor(nodeIds.length / 2) + 1,
    };

    setInterval(() => this.heartbeat(), this.heartbeatInterval);
  }

  /**
   * Write data with quorum and generation-based fencing
   */
```

```
  async write<T>(data: T): Promise<{ success: boolean; generation: number }> {
    // Check if we're leader and have valid lease
    if (!this.state.isLeader) {
      return { success: false, generation: this.state.generation };
    }

    if (Date.now() > this.state.leaseExpiry) {
      // Lease expired, stop being leader
      this.state.isLeader = false;
      return { success: false, generation: this.state.generation };
    }

    // Have quorum and valid lease, write to local store
    const dataId = this.generateId();
    this.replicaData.set(dataId, { data, generation: this.state.generation });

    // Replicate to other nodes with our generation
    const replicationResults = await this.replicateToQuorum(
      dataId,
      data,
      this.state.generation
    );

    // Check if we got quorum acks
    const ackCount = replicationResults.filter(r => r.success).length;
    if (ackCount >= this.state.quorumSize) {
      return { success: true, generation: this.state.generation };
    } else {
      // Lost quorum during write, rollback
      this.replicaData.delete(dataId);
      this.state.isLeader = false;
      return { success: false, generation: this.state.generation };
    }
  }

  /**
   * Replicate to quorum of nodes
   */
  private async replicateToQuorum(
    dataId: string,
    data: any,
    generation: number
  ): Promise<Array<{ success: boolean }>> {
    const healthyNodes = Array.from(this.nodes.values()).filter(n => n.healthy);

    const results = await Promise.all(
      healthyNodes.map(node => this.sendReplicateRpc(node.id, dataId, data,
generation))
    );

    return results;
  }
```

```
  /**
   * Handle replica RPC (from leader)
   */
  async handleReplicateRpc(
    leaderId: string,
    generation: number,
    dataId: string,
    data: any
  ): Promise<{ success: boolean }> {
    // Reject if from old leader (stale generation)
    if (generation < this.state.generation) {
      return { success: false };
    }

    // Accept if from current or newer generation
    if (generation > this.state.generation) {
      // Newer generation, step down as leader
      this.state.isLeader = false;
      this.state.generation = generation;
    }

    // Store the data
    this.replicaData.set(dataId, { data, generation });
    return { success: true };
  }

  /**
   * Heartbeat to detect failures and maintain leadership
   */
  private async heartbeat(): Promise<void> {
    // Update node health (ping all nodes)
    await this.updateNodeHealth();

    // Count healthy nodes
    const healthyCount = Array.from(this.nodes.values()).filter(n =>
n.healthy).length;

    // Check if we have quorum
    const hasQuorum = healthyCount >= this.state.quorumSize;

    if (!hasQuorum && this.state.isLeader) {
      // Lost quorum while being leader, step down
      console.log(`Lost quorum (${healthyCount}/${this.state.quorumSize}), stepping
down`);
      this.state.isLeader = false;
    }

    if (this.state.isLeader && Date.now() > this.state.leaseExpiry) {
      // Lease expired, need to renew or step down
      const renewalSuccess = await this.renewLease();
      if (!renewalSuccess) {
```

```
        this.state.isLeader = false;
      }
    }

    if (!this.state.isLeader && hasQuorum) {
      // We don't have leadership but have quorum, attempt to become leader
      const leadershipSuccess = await this.attemptLeadership();
      if (leadershipSuccess) {
        this.state.isLeader = true;
        this.state.generation++;
        this.state.leaseExpiry = Date.now() + this.leaseTimeout;
        console.log(`Became leader with generation ${this.state.generation}`);
      }
    }
  }

  /**
   * Update health status of all nodes
   */
  private async updateNodeHealth(): Promise<void> {
    const healthChecks = Array.from(this.nodes.keys()).map(nodeId =>
      this.pingNode(nodeId)
        .then(() => ({ nodeId, healthy: true }))
        .catch(() => ({ nodeId, healthy: false }))
    );

    const results = await Promise.allSettled(healthChecks);

    for (const result of results) {
      if (result.status === 'fulfilled') {
        const { nodeId, healthy } = result.value;
        this.nodes.set(nodeId, { id: nodeId, healthy });
      }
    }
  }

  /**
   * Attempt to become leader (requires quorum acknowledgment)
   */
  private async attemptLeadership(): Promise<boolean> {
    const healthyNodes = Array.from(this.nodes.values()).filter(n => n.healthy);

    const voteResults = await Promise.all(
      healthyNodes.map(node =>
        this.requestVote(node.id, this.state.generation + 1)
          .then(success => success)
          .catch(() => false)
      )
    );

    const voteCount = voteResults.filter(v => v).length;
    return voteCount >= this.state.quorumSize;
```

```
}

/**
 * Renew leadership lease
 */
private async renewLease(): Promise<boolean> {
  const healthyNodes = Array.from(this.nodes.values()).filter(n => n.healthy);

  const renewalResults = await Promise.all(
    healthyNodes.map(node =>
      this.sendHeartbeat(node.id, this.state.generation)
        .then(() => true)
        .catch(() => false)
    )
  );

  const renewalCount = renewalResults.filter(r => r).length;
  const hasQuorum = renewalCount >= this.state.quorumSize;

  if (hasQuorum) {
    this.state.leaseExpiry = Date.now() + this.leaseTimeout;
  }

  return hasQuorum;
}

/**
 * RPC: Ping node for health check
 */
private async pingNode(nodeId: string): Promise<void> {
  // Simulated RPC
  const node = this.nodes.get(nodeId);
  if (!node) throw new Error('Node not found');
  if (!node.healthy) throw new Error('Node unhealthy');
}

/**
 * RPC: Send heartbeat to replica
 */
private async sendHeartbeat(nodeId: string, generation: number): Promise<void> {
  // Simulated RPC
}

/**
 * RPC: Request vote for leadership
 */
private async requestVote(nodeId: string, generation: number): Promise<boolean> {
  // Simulated RPC (would check if we can vote for this generation)
  return true;
}

/**
```

```
   * RPC: Send replicate request
   */
  private async sendReplicateRpc(
    nodeId: string,
    dataId: string,
    data: any,
    generation: number
  ): Promise<{ success: boolean }> {
    // Simulated RPC
    return { success: true };
  }

  /**
   * Generate unique ID
   */
  private generateId(): string {
    return `${Date.now()}-${Math.random().toString(36).substr(2, 9)}`;
  }

  /**
   * Get system status
   */
  getStatus() {
    const healthyCount = Array.from(this.nodes.values()).filter(n =>
n.healthy).length;
    return {
      isLeader: this.state.isLeader,
      generation: this.state.generation,
      healthyNodes: healthyCount,
      quorumSize: this.state.quorumSize,
      hasQuorum: healthyCount >= this.state.quorumSize,
      leaseValid: Date.now() < this.state.leaseExpiry,
    };
  }
}

// Example usage with network partition simulation
class NetworkPartitionExample {
  private system: DistributedSystem;

  constructor() {
    this.system = new DistributedSystem(['A', 'B', 'C']);
  }

  async simulatePartition() {
    console.log('=== Initial state ===');
    console.log(this.system.getStatus());
    // Output: { isLeader: true, generation: 1, healthyNodes: 3, quorumSize: 2,
hasQuorum: true }

    console.log('\n=== Network partition: A isolated ===');
    // Simulated: Node A loses network connectivity
```

```
    // (In real implementation: A cannot reach B or C)

    // Node A: healthyNodes = 1, quorumSize = 2, no quorum
    // Node A steps down as leader

    // Node B: healthyNodes = 2, quorumSize = 2, has quorum
    // Node C: healthyNodes = 2, quorumSize = 2, has quorum
    // B and C elect new leader

    console.log('\n=== B (majority partition) becomes leader ===');
    // B can now accept writes
    // A cannot accept writes (no quorum)

    console.log('\n=== Network partition heals ===');
    // A can reach B and C again
    // A discovers B is leader with higher generation
    // A catches up from B's data
    // System consistent again
  }
}

// Express API with split brain prevention
const system = new DistributedSystem(['node1', 'node2', 'node3']);

app.post('/write', async (req, res) => {
  const result = await system.write(req.body);

  if (result.success) {
    res.status(200).json({
      success: true,
      generation: result.generation,
      message: 'Data written to quorum',
    });
  } else {
    res.status(503).json({
      success: false,
      message: 'Cannot write: not leader or lost quorum',
      status: system.getStatus(),
    });
  }
});

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

app.post('/admin/simulate-partition', (req, res) => {
  // Simulate network partition for testing
  const example = new NetworkPartitionExample();
  example.simulatePartition();
```

```
  res.json({ message: 'Partition simulation complete' });
});
```

6. Correct Usage Patterns (Where This Shines)

Pattern 1: Leader-Based Database Replication

```
// Multiple data centers, automatic failover
// Only quorum partition can accept writes
// Minority partition blocks writes automatically

if (system.hasQuorum) {
  // Attempt leadership
  system.write(data);
} else {
  // Cannot write, wait for partition to heal
  return 503 'Service Unavailable';
}
```

Pattern 2: Distributed Configuration Service

```
// Config changes only accepted by leader with quorum
// Clients read from any replica
// Config version number acts as generation

leader.updateConfig(newConfig, generation);
// Replicates to quorum with generation number
// Old config version rejected

follower.updateConfig(newConfig, oldGeneration);
// Rejected: oldGeneration < currentGeneration
```

Pattern 3: Lease-Based Primary Election

```
// Primary holds exclusive lease
// Lease must be renewed every N seconds
// If primary dies, lease expires
// Another node becomes primary after lease expiry

if (lease.isValid) {
  acceptWrites();
}
// After lease expires, leader steps down
```

7. Failure Modes & Edge Cases

Byzantine Failure (Malicious Node)

Problem: One node deliberately sends conflicting information.

Mitigation: Use Byzantine fault tolerance (Raft/Paxos not sufficient).

Cascading Partitions

Problem: Network partitions multiple times during recovery.

Symptoms:

- A-B partition heals
- B becomes leader
- A-B reconnect, A catches up
- Then A-C partition occurs
- C thinks A + C has quorum (2/3)
- C becomes leader during reconnection
- Two leaders possible

Mitigation: Increase lease/heartbeat frequency, use epoch-based authority.

Slow Network vs Partition

Problem: Network very slow (high latency) vs network actually down.

Symptoms:

- Node detects other unreachable
- But other node still healthy (just slow to respond)
- Both nodes think other is dead
- Both attempt leadership

Mitigation: Distinguish between timeout and explicit failure, use multiple heartbeat channels.

Asymmetric Partitions

Problem: Network partition is one-way (A→B works, B→A fails).

Symptoms:

- A can reach B, thinks B is healthy
- B cannot reach A, thinks A is dead
- Inconsistent view of cluster

Mitigation: Require bidirectional communication verification.

8. Performance Characteristics & Tradeoffs

Availability vs Consistency

No partition handling:

- Higher availability (any node can be primary)
- Lower consistency (split brain possible)

Quorum-based partition handling:

- Lower availability during partition (minority partition unavailable)

- Higher consistency (no split brain)

Latency

Quorum writes:

- Must wait for majority quorum to acknowledge
- Higher latency than single-node write
- Trade-off: Safety vs speed

Complexity

No partition handling:

- Simple to implement
- But data corruption possible

Partition-aware:

- More complex (quorum logic, fencing, generation)
- But prevents data corruption

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

Mistake 1: Quorum Check Without Enforcement

Fix: Actively reject writes if no quorum.

Mistake 2: Forgetting to Increment Generation

Fix: Always increment generation on leadership change.

Mistake 3: No Fencing of Old Leader

Fix: Implement lease or generation-based fencing.

Mistake 4: Assuming Clean Partitions

Fix: Handle asymmetric and cascading partitions.

Mistake 5: No Monitoring of Partition Events

Fix: Alert on leadership changes, quorum loss.

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

Anti-Pattern 1: Over-Reliance on Timeouts

Don't assume timeout = partition (could be slow network).

Anti-Pattern 2: Ignoring Byzantine Faults

Quorum doesn't protect against malicious nodes.

Anti-Pattern 3: Too-Strict Quorum

N=3, quorum=2: Any 2-node failure = system down.

11. Related Concepts (With Contrast)

Consensus Algorithms (Raft/Paxos)

Difference: Consensus handles arbitrary network partitions. Quorum-based assumes majority/minority split.

Distributed Transactions

Difference: Transactions coordinate writes across nodes. Quorum prevents conflicting writes from multiple leaders.

Eventual Consistency

Difference: Eventual consistency accepts temporary divergence. Quorum prevents divergence from occurring.

12. Production Readiness Checklist

Quorum Configuration

- [ ] Set cluster size odd (3, 5, 7)
- [ ] Calculate quorum size correctly
- [ ] Configure for each cluster tier

Leadership

- [ ] Implement leader election
- [ ] Include generation/epoch numbers
- [ ] Lease-based fencing implemented

Monitoring

- [ ] Alert on leadership change
- [ ] Alert on quorum loss
- [ ] Track partition events
- [ ] Dashboard showing cluster health

Testing

- [ ] Simulate network partitions
- [ ] Verify minority partition blocks writes
- [ ] Test partition healing
- [ ] Verify no data divergence

Documentation

- [ ] Document quorum strategy
- [ ] Document fencing mechanism
- [ ] Runbook for partition recovery