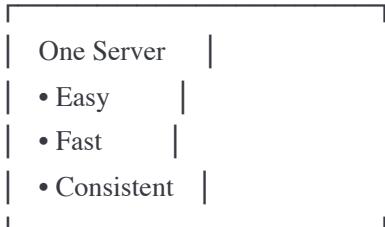


Chapter 16: Distributed Systems Theory

Introduction: The Challenges of Distribution

When you distribute a system across multiple machines, fundamental challenges emerge.

Single Machine (Simple):



Distributed System (Complex):

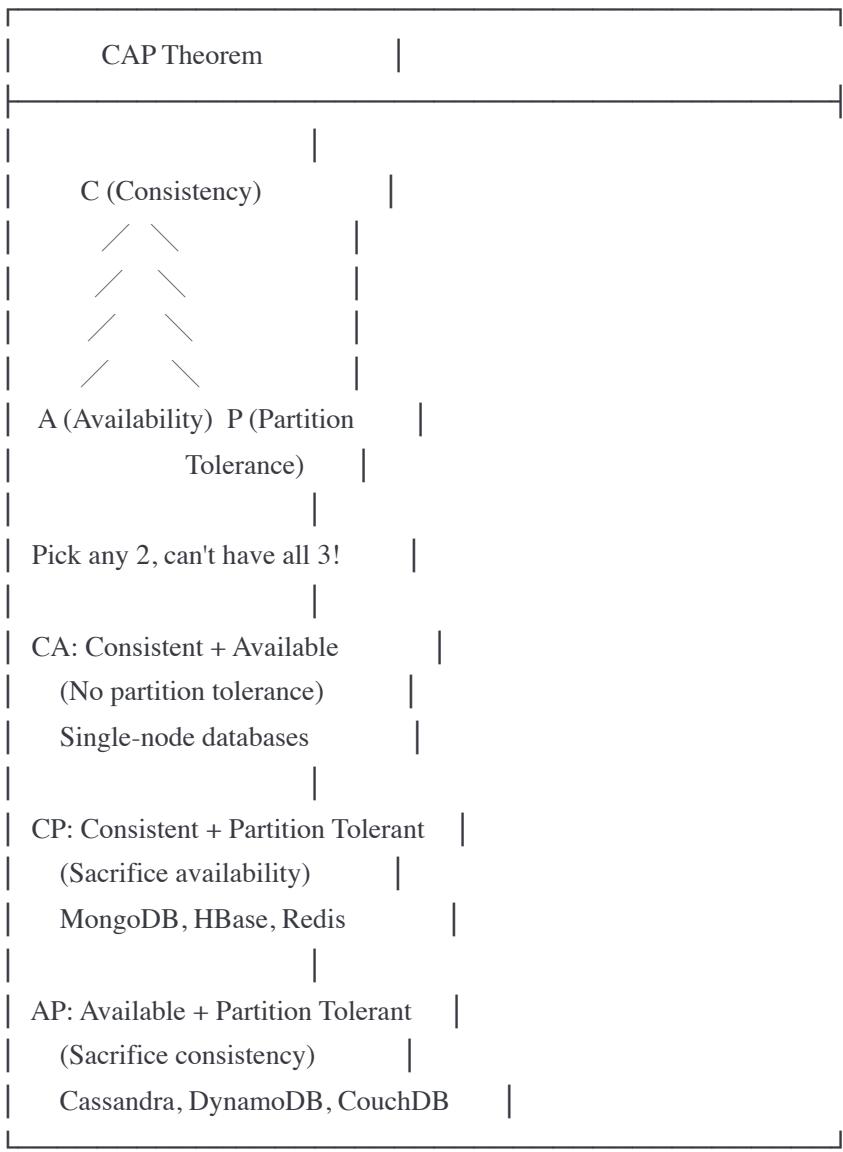


Challenges:

- Network can fail (partition)
- Network has latency
- Clocks are not synchronized
- Machines can fail independently
- Data must be coordinated

1. CAP Theorem

CAP Theorem (Brewer's Theorem): You can only have 2 out of 3 properties in a distributed system.



C - Consistency

Definition: All nodes see the same data at the same time.

Consistent System:

Time	Node A	Node B	Node C
------	--------	--------	--------

10:00	balance: \$100	balance: \$100	balance: \$100
-------	----------------	----------------	----------------

10:01	(write: -\$50)		
-------	----------------	--	--

	balance: \$50	balance: \$50	balance: \$50
--	---------------	---------------	---------------

↑ Immediately synchronized

All nodes always have same value!

Inconsistent System:

Time	Node A	Node B	Node C
------	--------	--------	--------

10:00	balance: \$100	balance: \$100	balance: \$100
-------	----------------	----------------	----------------

10:01	(write: -\$50)		
-------	----------------	--	--

	balance: \$50	balance: \$100	balance: \$100
--	---------------	----------------	----------------

↑ Not yet synchronized

10:02		balance: \$50	balance: \$50
-------	--	---------------	---------------

↑ Eventually synchronized

Nodes temporarily have different values!

Real-World Example:

Banking System (Must be Consistent):

User checks balance on Node A: \$1000

User tries to withdraw \$900 on Node B

If not consistent:

- Node A: balance = \$1000
- Node B: balance = \$1000 (stale data)
- Both approve withdrawal!
- Total withdrawn: \$1800 from \$1000 account!
- ✗ DISASTER!

With consistency:

- Node A: balance = \$1000
- Write: balance = \$100 (after first withdrawal)
- Node B: balance = \$100 (synchronized immediately)
- Second withdrawal rejected!
- ✓ Correct behavior

A - Availability

Definition: Every request gets a response (success or failure).

Available System:

User Request → Server



Response (always returned)

Even if:

- Data is stale
- Server is overloaded
- Network is slow
- Still returns something

Unavailable System:

User Request → Server



No response (timeout)

or

Error (service down)

Example:

Social Media Feed (Can sacrifice consistency for availability):

User posts photo

- |— Write to Node A: Success ✓
- |— Replicate to Node B: In progress...
- |— Replicate to Node C: In progress...

Friend views feed (reads from Node B):

- |— Node B doesn't have photo yet
- |— But still returns response (old data)
- |— Photo appears in ~100ms when replication completes

Trade-off: Availability (always get response)

over

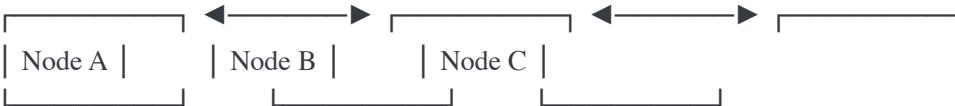
Consistency (might not see latest)

P - Partition Tolerance

Definition: System continues working even when network splits.

Network Partition:

Normal:



Partition (Network failure):



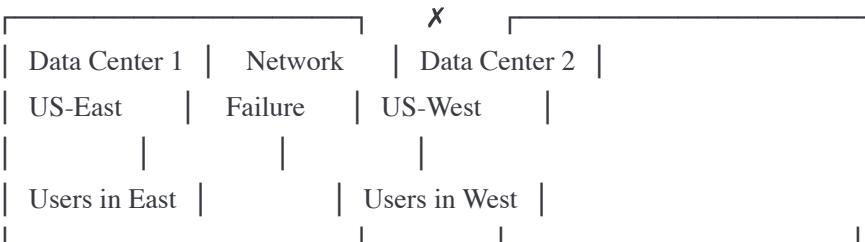
Isolated! Can communicate

Partition-tolerant system:

- Continues operating despite partition
- Nodes A operates independently
- Nodes B+C operate together
- Eventually reconcile when network heals

Real-World Scenario:

Data Center Network Partition:



Without Partition Tolerance:

- Entire system goes down
- All users affected

With Partition Tolerance:

- East serves East users (maybe with stale data)
- West serves West users (maybe with stale data)
- When network heals, reconcile differences
- Users still served (availability maintained)

CAP Theorem in Practice

You MUST choose:

Scenario: Network Partition Occurs

Choice 1: CP (Consistency + Partition Tolerance)

Reject writes during partition

→ Guarantees consistency

→ But sacrifices availability

Example: MongoDB

If can't reach majority → reject writes

Choice 2: AP (Availability + Partition Tolerance)

Accept writes on both sides of partition

→ Guarantees availability

→ But data may diverge (inconsistent)

Example: Cassandra, DynamoDB

Each side accepts writes

Reconcile later (conflict resolution)

Can't have CA:

Network partitions WILL happen (it's reality)

So you can't ignore P

Must choose between C and A

Database Classification

Database	Type	Use Case
----------	------	----------

PostgreSQL	CA	Single node
------------	----	-------------

MySQL (single node)	CA	Strong ACID
------------------------	----	-------------

MongoDB	CP	Financial data
---------	----	----------------

HBase	CP	Consistency	
Redis	CP	critical	
Cassandra	AP	High	
DynamoDB	AP	availability	
Riak	AP	Always on	

Note: Modern databases often allow tuning!

Example: Cassandra can be configured for CP or AP

CAP Theorem Simulation

javascript

```

class DistributedSystem {
  constructor(nodes) {
    this.nodes = nodes;
    this.networkPartitioned = false;
  }

  // CP System: Consistent + Partition Tolerant
  async writeCP(key, value) {
    if (this.networkPartitioned) {
      // Can't guarantee consistency during partition
      // Reject write (sacrifice availability)
      throw new Error('Cannot write during partition (CP mode)');
    }

    // Write to all nodes (strong consistency)
    for (const node of this.nodes) {
      await node.write(key, value);
    }

    console.log(`CP Write: ${key} = ${value} (all nodes updated)`);
  }

  async readCP(key) {
    if (this.networkPartitioned) {
      throw new Error('Cannot read during partition (CP mode)');
    }

    // Read from any node (all have same data)
    return await this.nodes[0].read(key);
  }
}

// AP System: Available + Partition Tolerant
async writeAP(key, value) {
  const reachableNodes = this.getReachableNodes();

  if (reachableNodes.length === 0) {
    throw new Error('No nodes reachable');
  }

  // Write to reachable nodes only
  for (const node of reachableNodes) {
    await node.write(key, value);
  }

  console.log(`AP Write: ${key} = ${value} (${reachableNodes.length} nodes updated)`);
}

```

```

// Accept write even during partition!
// Availability maintained, but may be inconsistent
}

async readAP(key) {
  const reachableNodes = this.getReachableNodes();

  if (reachableNodes.length === 0) {
    throw new Error('No nodes reachable');
  }

  // Read from any reachable node
  // May return stale data during partition
  return await reachableNodes[0].read(key);
}

getReachableNodes() {
  // During partition, only some nodes reachable
  if (this.networkPartitioned) {
    return this.nodes.slice(0, Math.floor(this.nodes.length / 2));
  }
  return this.nodes;
}

simulatePartition() {
  console.log('🔥 Network partition occurred!');
  this.networkPartitioned = true;
}

healPartition() {
  console.log('✅ Network partition healed!');
  this.networkPartitioned = false;
}

// Usage
const nodes = [
  new Node('A'),
  new Node('B'),
  new Node('C')
];

const system = new DistributedSystem(nodes);

// Normal operation
await system.writeCP('balance', 1000); // ✅ Success
await system.readCP('balance');      // → 1000

```

```

// Network partition occurs
system.simulatePartition();

// CP System: Rejects operations
try {
    await system.writeCP('balance', 500); // ✗Throws error
} catch (error) {
    console.log('CP: Cannot write during partition');
}

// AP System: Accepts operations (may be inconsistent)
await system.writeAP('balance', 500); // ✓Success (partial nodes)
await system.readAP('balance');      // → 500 (from reachable nodes)

// Other partition might still see 1000!
// Inconsistent but available

```

2. BASE vs ACID

ACID (Traditional Databases)

Already covered in Chapter 6, quick recap:

ACID = Strong Consistency Guarantees

A - Atomicity: All or nothing

C - Consistency: Valid state always

I - Isolation: Transactions don't interfere

D - Durability: Changes survive crashes

Example:

BEGIN TRANSACTION;

UPDATE account SET balance = balance - 100 WHERE id = 1;

UPDATE account SET balance = balance + 100 WHERE id = 2;

COMMIT;

Either both succeed or both fail

Strong guarantees!

BASE (Distributed Systems)

BASE = Weak Consistency, High Availability

B - Basically Available

System available most of the time

May return stale data

A - Soft state

State may change without input (due to eventual consistency)

S - Eventual consistency

System becomes consistent over time

No guarantees when

BASE Example

Social Media Post:

User A posts: "Hello world!"

 |— Writes to Node 1 (US-East)

 |— Status: Success ✓

 |— Replicates to Node 2 (EU-West)

 |— Status: In progress...

 |— Replicates to Node 3 (Asia)

 |— Status: In progress...

User B in Europe (reads from Node 2):

Time 0ms: Post not visible yet

Time 50ms: Post appears!

User C in Asia (reads from Node 3):

Time 0ms: Post not visible yet

Time 200ms: Post appears!

Basically Available: All users got response

Soft State: Post visibility changed (without input)

Eventual Consistency: Eventually all see the post

ACID vs BASE Comparison

Property	ACID	BASE	
Consistency	Strong	Eventual	

Availability	Lower	Higher	
Partition	Not tolerant	Tolerant	
Performance	Slower	Faster	
Scalability	Limited	Excellent	
Complexity	Lower	Higher	
<hr/>			
Best For	Financial	Social media	
	Inventory	Analytics	
	Bookings	Caching	
	Logs		

Trade-off:

ACID: Correct but slow, limited scale

BASE: Fast and scalable, but eventually consistent

3. Eventual Consistency

What is Eventual Consistency?

Definition: If no new updates, all replicas will eventually have the same data.

Timeline of Eventual Consistency:

T=0ms: Write to Node A (value = 100)

Node A: 100

Node B: 0 (old)

Node C: 0 (old)

↓ Inconsistent!

T=50ms: Replication to Node B

Node A: 100

Node B: 100

Node C: 0 (old)

↓ Still inconsistent!

T=200ms: Replication to Node C

Node A: 100

Node B: 100

Node C: 100

↓ Eventually consistent!

Key Point: No guarantee WHEN consistency achieved

But guarantee it WILL be achieved

Eventual Consistency Challenges

Problem 1: Read Your Own Writes

User posts comment:

T=0ms: POST /comments → Write to Node A

Server: "Comment created!"

T=10ms: GET /comments → Read from Node B (faster, closer)

Node B: Doesn't have new comment yet!

User sees: Comment disappeared! 

Solution: Read from same node you wrote to (session stickiness)

Implementation:

javascript

```
class EventuallyConsistentDB {
  constructor() {
    this.nodes = {
      'node-a': new Map(),
      'node-b': new Map(),
      'node-c': new Map()
    };
    this.replicationDelay = 100; // ms
  }

  async write(key, value, nodeId = 'node-a') {
    // Write to primary node
    this.nodes[nodeId].set(key, {
      value,
      timestamp: Date.now(),
      version: (this.nodes[nodeId].get(key)?.version || 0) + 1
    });

    console.log(`Written to ${nodeId}: ${key} = ${value}`);

    // Async replication (eventual)
    this.replicateAsync(key, value, nodeId);

    return { nodeId, version: this.nodes[nodeId].get(key).version };
  }

  async replicateAsync(key, value, sourceNode) {
    // Simulate network delay
    setTimeout(() => {
      const sourceData = this.nodes[sourceNode].get(key);

      for (const nodeId in this.nodes) {
        if (nodeId !== sourceNode) {
          const targetData = this.nodes[nodeId].get(key);

          // Only update if newer
          if (!targetData || sourceData.version > targetData.version) {
            this.nodes[nodeId].set(key, { ...sourceData });
            console.log(`Replicated to ${nodeId}: ${key} = ${value}`);
          }
        }
      }
    }, this.replicationDelay);
  }

  async read(key, nodeId = 'node-b') {
```

```

const data = this.nodes[nodeId].get(key);

if (!data) {
    return null;
}

console.log(`Read from ${nodeId}: ${key} = ${data.value}`);
return data.value;
}

async readYourWrites(key, writeNodeId) {
    // Read from same node you wrote to
    return await this.read(key, writeNodeId);
}
}

// Demo
const db = new EventuallyConsistentDB();

// Write
const writeResult = await db.write('balance', 1000);
console.log('Write completed');

// Read immediately from different node
const value1 = await db.read('balance', 'node-b');
console.log('Immediate read:', value1); // → null or old value

// Wait for replication
await new Promise(r => setTimeout(r, 150));

// Read again
const value2 = await db.read('balance', 'node-b');
console.log('After replication:', value2); // → 1000

// Solution: Read from same node
const value3 = await db.readYourWrites('balance', writeResult.nodeId);
console.log('Read your writes:', value3); // → 1000 (immediate)

```

Problem 2: Concurrent Writes (Conflicts)

Conflict Scenario:

User A: balance = \$1000

User B: balance = \$1000

T=0ms: User A withdraws \$500 (writes to Node 1)

Node 1: balance = \$500

T=0ms: User B withdraws \$600 (writes to Node 2)

Node 2: balance = \$400

T=100ms: Replication occurs

Conflict! Which value is correct?

Node 1: \$500

Node 2: \$400

Conflict Resolution Strategies:

1. Last Write Wins (LWW)
2. Vector Clocks
3. Application-specific logic

Last Write Wins Implementation:

```
javascript
```

```

class LWWDatabase {
  constructor() {
    this.data = new Map();
  }

  write(key, value, timestamp) {
    const existing = this.data.get(key);

    if (!existing || timestamp > existing.timestamp) {
      // Newer write wins
      this.data.set(key, { value, timestamp });
      console.log(`LWW: Accepted write ${key}=${value} (ts: ${timestamp})`);
      return true;
    } else {
      // Older write discarded
      console.log(`LWW: Rejected write ${key}=${value} (ts: ${timestamp}) - older than ${existing.timestamp}`);
      return false;
    }
  }

  read(key) {
    return this.data.get(key)?.value;
  }
}

// Simulation
const db = new LWWDatabase();

// User A writes (timestamp: 1000)
db.write('balance', 500, 1000);

// User B writes (timestamp: 1001 - later)
db.write('balance', 400, 1001);

console.log('Final value:', db.read('balance')); // → 400 (LWW)

// Problem: User A's withdrawal lost!
// $500 + $600 = $1100 withdrawn from $1000 account

```

Vector Clocks (Detecting Conflicts)

Vector Clock: Track causality

Node A: [A:1, B:0, C:0] (Node A made 1 update)

Node B: [A:1, B:1, C:0] (Knows about A's update, made 1 update)

Example:

T=0: Initial: [A:0, B:0, C:0] balance=\$1000

T=1: Node A writes \$500

Clock: [A:1, B:0, C:0]

T=2: Node B writes \$400 (hasn't seen A's write)

Clock: [A:0, B:1, C:0]

T=3: Replication occurs

Node A has: [A:1, B:0, C:0] = \$500

Node B has: [A:0, B:1, C:0] = \$400

Compare clocks:

Neither is strictly newer!

→ Conflict detected!

→ Requires resolution

Implementation:

javascript

```
class VectorClock {
  constructor(nodeId, nodes) {
    this.nodeId = nodeId;
    this.clock = {};

    // Initialize clock
    nodes.forEach(node => {
      this.clock[node] = 0;
    });
  }

  increment() {
    this.clock[this.nodeId]++;
  }

  update(otherClock) {
    // Merge clocks (take max for each node)
    for (const node in otherClock) {
      this.clock[node] = Math.max(
        this.clock[node] || 0,
        otherClock[node] || 0
      );
    }
  }

  compare(otherClock) {
    // Returns: 'before', 'after', 'concurrent'

    let before = false;
    let after = false;

    for (const node in this.clock) {
      if (this.clock[node] < (otherClock[node] || 0)) {
        before = true;
      }
      if (this.clock[node] > (otherClock[node] || 0)) {
        after = true;
      }
    }

    if (before && after) {
      return 'concurrent'; // Conflict!
    } else if (before) {
      return 'before';
    } else if (after) {
      return 'after';
    }
  }
}
```

```

} else {
    return 'equal';
}
}

clone() {
    const cloned = new VectorClock(this.nodeId, Object.keys(this.clock));
    cloned.clock = { ...this.clock };
    return cloned;
}
}

// Usage
const clockA = new VectorClock('A', ['A', 'B', 'C']);
const clockB = new VectorClock('B', ['A', 'B', 'C']);

// Node A makes update
clockA.increment(); // [A:1, B:0, C:0]

// Node B makes concurrent update (hasn't seen A's update)
clockB.increment(); // [A:0, B:1, C:0]

// Detect conflict
const comparison = clockA.compare(clockB.clock);
console.log('Comparison:', comparison); // → 'concurrent' (conflict!)

if (comparison === 'concurrent') {
    console.log('⚠️ Conflict detected! Need resolution');
    // Application must decide how to merge
}

```

4. Strong vs Eventual Consistency Trade-offs

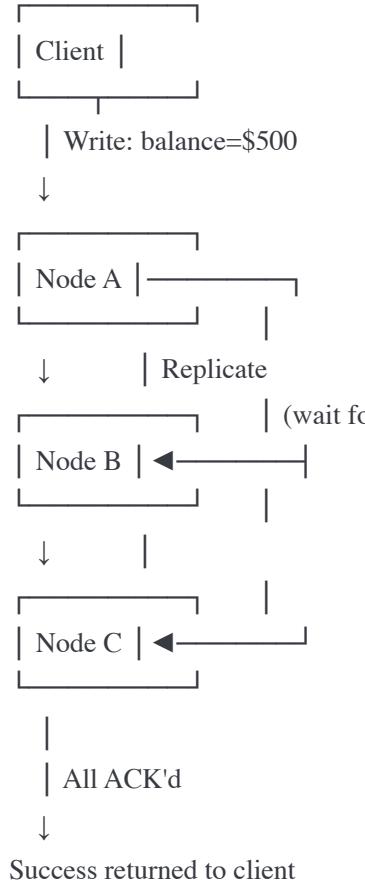
Strong Consistency

Definition: All reads return the most recent write.

Strong Consistency Implementation:

Write Process:

1. Client writes to Node A
2. Node A replicates to B, C
3. Wait for acknowledgment from ALL nodes
4. Only then return success to client



Latency: High (wait for all nodes)

Consistency: Perfect

Availability: Lower (if node down, write fails)

Implementation:

javascript

```
class StronglyConsistentDB {
  constructor(nodes) {
    this.nodes = nodes;
    this.quorum = Math.floor(nodes.length / 2) + 1;
  }

  async write(key, value) {
    console.log(`Writing ${key}=${value} to ${this.nodes.length} nodes...`);

    const promises = this.nodes.map(node =>
      node.write(key, value)
    );

    try {
      // Wait for ALL nodes to acknowledge
      await Promise.all(promises);

      console.log(`✓ Write successful (all nodes acknowledged)`);
      return { success: true };
    } catch (error) {
      // If any node fails, entire write fails
      console.log(`✗ Write failed (node unavailable)`);
      throw error;
    }
  }
}

async read(key) {
  // Read from majority (quorum)
  const promises = this.nodes.slice(0, this.quorum).map(node =>
    node.read(key)
  );

  const results = await Promise.all(promises);

  // All should have same value (strong consistency)
  const value = results[0]!.value;

  // Verify consistency
  const allSame = results.every(r => r!.value === value);

  if (!allSame) {
    console.warn(`⚠️ Inconsistency detected!`);
  }

  return value;
}
```

```
}

}

// Usage
const db = new StronglyConsistentDB([nodeA, nodeB, nodeC]);

// Write (slow but consistent)
await db.write('balance', 1000); // Waits for all 3 nodes

// Read
const balance = await db.read('balance'); // → Always latest value

// If a node is down:
nodeB.healthy = false;
await db.write('balance', 500); // → Throws error (can't reach all nodes)
```

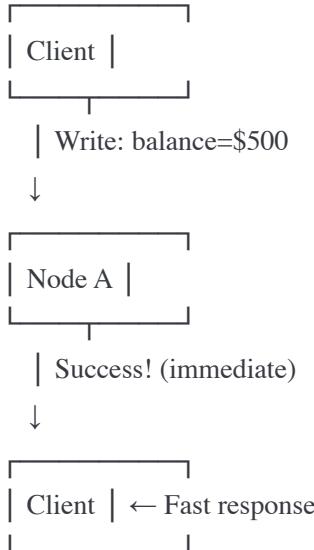
Eventual Consistency

Definition: Replicas may differ temporarily, but converge eventually.

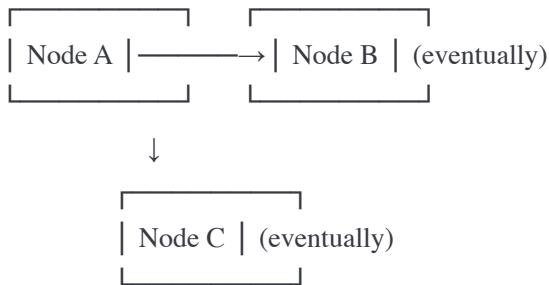
Eventual Consistency Implementation:

Write Process:

1. Client writes to nearest node
2. Node returns success immediately
3. Async replication to other nodes
4. Eventually all nodes have data



Meanwhile (async):



Latency: Low (don't wait for replication)

Consistency: Eventual

Availability: High (single node can handle write)

Implementation:

javascript

```
class EventuallyConsistentDB {
  constructor(nodes) {
    this.nodes = nodes;
    this.replicationQueue = [];
  }

  async write(key, value, preferredNode = 0) {
    const node = this.nodes[preferredNode];

    // Write to single node
    const version = await node.write(key, value);

    console.log(`✅ Write successful (Node ${preferredNode})`);

    // Queue async replication (don't wait!)
    this.queueReplication(key, value, version, preferredNode);

    // Return immediately
    return { success: true, node: preferredNode };
  }

  queueReplication(key, value, version, sourceNode) {
    // Replicate asynchronously
    setTimeout(async () => {
      console.log(`Replicating ${key}=${value} to other nodes...`);

      for (let i = 0; i < this.nodes.length; i++) {
        if (i !== sourceNode) {
          try {
            await this.nodes[i].write(key, value, version);
            console.log(`Replicated to Node ${i}`);
          } catch (error) {
            console.error(`Replication to Node ${i} failed:`, error);
            // Retry later
          }
        }
      }
    }, 0); // Async
  }

  async read(key, preferredNode = null) {
    // Read from nearest/fastest node
    const nodeIndex = preferredNode ?? Math.floor(Math.random() * this.nodes.length);
    const node = this.nodes[nodeIndex];

    const result = await node.read(key);
```

```

console.log(`Read from Node ${nodeIndex}: ${key} = ${result?.value}`);

return result?.value;
}

async readWithReadRepair(key) {
    // Read from all nodes
    const promises = this.nodes.map(node => node.read(key));
    const results = await Promise.all(promises);

    // Find latest version
    const latest = results.reduce((max, curr) => {
        if (!max || (curr && curr.version > max.version)) {
            return curr;
        }
        return max;
    }, null);

    // Repair stale nodes
    for (let i = 0; i < results.length; i++) {
        if (results[i]?.version < latest.version) {
            console.log(`Repairing Node ${i} with latest value`);
            await this.nodes[i].write(key, latest.value, latest.version);
        }
    }

    return latest?.value;
}

// Usage
const db = new EventuallyConsistentDB([nodeA, nodeB, nodeC]);

// Fast write
await db.write('balance', 1000, 0); // Immediate response!

// Read immediately (might get stale data)
const value1 = await db.read('balance', 1);
// → undefined or old value

// Wait for replication
await new Promise(r => setTimeout(r, 150));

// Now consistent
const value2 = await db.read('balance', 1);
// → 1000

```

Tunable Consistency (Cassandra)

Concept: Choose consistency level per operation!

Cassandra Consistency Levels:

WRITE Consistency:

Level	Replicas	Latency	Consistency
ONE	1 node ACK	Fast	Weak
QUORUM	Majority ACK	Medium	Strong
ALL	All nodes ACK	Slow	Strongest

READ Consistency:

Level	Replicas	Latency	Consistency
ONE	1 node	Fast	Weak
QUORUM	Majority	Medium	Strong
ALL	All nodes	Slow	Strongest

Strong Consistency = (Write_Quorum + Read_Quorum) > Replication_Factor

Example (Replication Factor = 3):

Write QUORUM (2) + Read QUORUM (2) = 4 > 3 ✓ Strong!

Write ONE (1) + Read ONE (1) = 2 < 3 ✗ Eventual

Choose per operation!

Implementation:

javascript

```

const cassandra = require('cassandra-driver');

const client = new cassandra.Client({
  contactPoints: ['node1', 'node2', 'node3'],
  localDataCenter: 'datacenter1',
  keyspace: 'myapp'
});

// Strong consistency (QUORUM)
async function writeStrong(userId, balance) {
  const query = 'UPDATE accounts SET balance = ? WHERE user_id = ?';

  await client.execute(query, [balance, userId], {
    consistency: cassandra.types.consistencies.quorum,
    prepare: true
  });

  console.log('Strong write completed (quorum acknowledged)');
}

async function readStrong(userId) {
  const query = 'SELECT balance FROM accounts WHERE user_id = ?';

  const result = await client.execute(query, [userId], {
    consistency: cassandra.types.consistencies.quorum
  });

  return result.rows[0].balance;
}

// Eventual consistency (ONE)
async function writeEventual(userId, viewCount) {
  const query = 'UPDATE page_views SET count = ? WHERE user_id = ?';

  await client.execute(query, [viewCount, userId], {
    consistency: cassandra.types.consistencies.one, // Fast!
    prepare: true
  });

  console.log('Eventual write completed (one node acknowledged)');
}

// Usage
// Critical data (balance): Strong consistency
await writeStrong(123, 1000);
const balance = await readStrong(123); // Guaranteed latest

```

```
// Non-critical data (views): Eventual consistency
await writeEventual(123, 1523); // Fast, don't wait for replication
```

5. Distributed Transactions and Two-Phase Commit

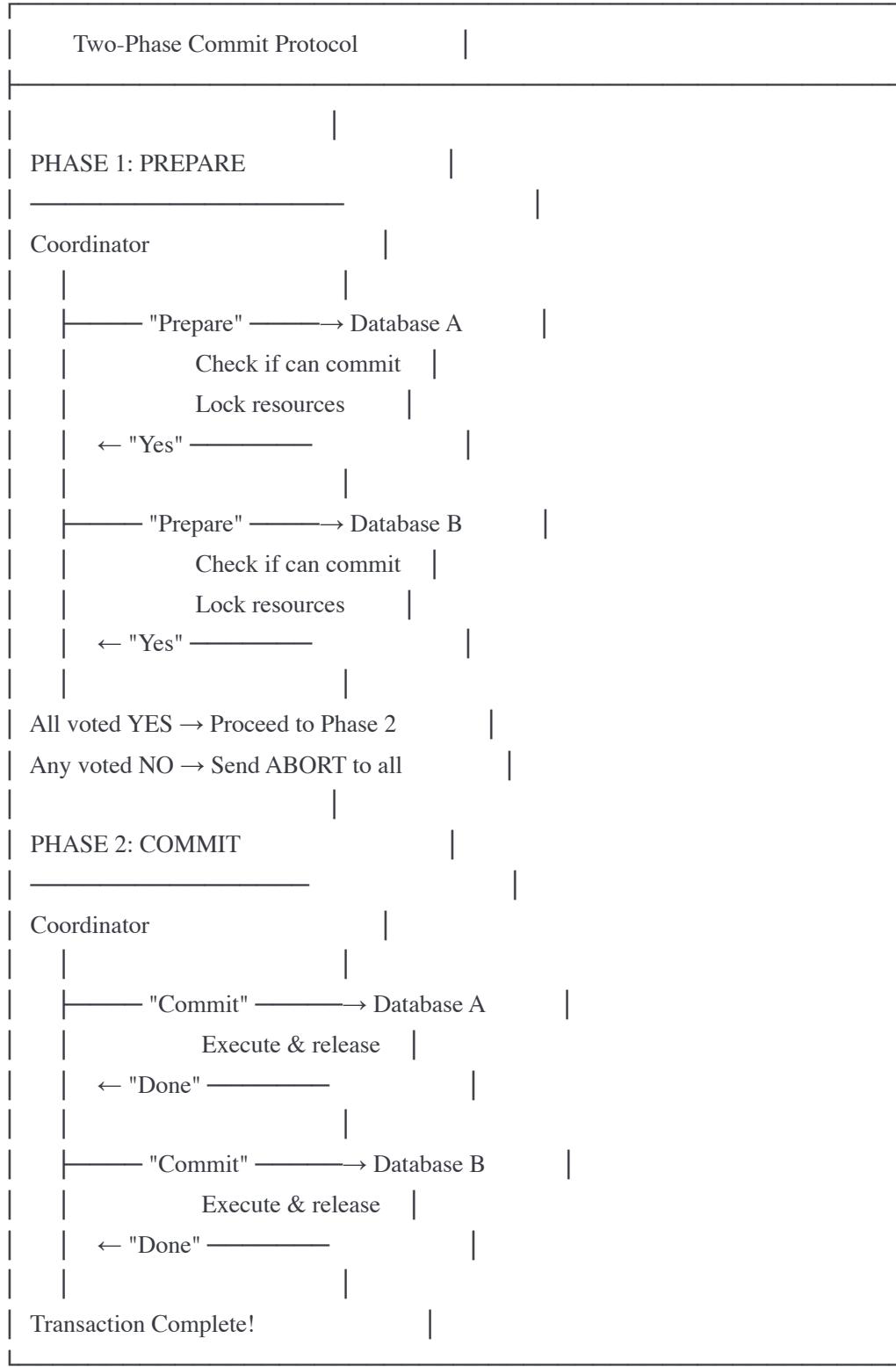
Two-Phase Commit (2PC)

Goal: Atomic commit across multiple databases.

Scenario: Transfer \$100 from Bank A to Bank B

Participants:

- Database A (Bank A's database)
- Database B (Bank B's database)
- Coordinator (manages transaction)



2PC Implementation

javascript

```
class TwoPhaseCommitCoordinator {
  constructor(participants) {
    this.participants = participants;
    this.transactionLog = [];
  }

  async executeTransaction(transactionId, operations) {
    console.log(`\n${'='.repeat(50)}`);
    console.log(` Starting 2PC: ${transactionId}`);
    console.log(`-${'='.repeat(50)}`);

    const context = {
      transactionId,
      operations,
      preparedParticipants: [],
      status: 'STARTED'
    };

    try {
      // PHASE 1: PREPARE
      console.log(`\n[PHASE 1] PREPARE`);
      console.log(`-${'='.repeat(50)}`);

      const preparePromises = this.participants.map((participant, idx) =>
        this.prepare(participant, operations[idx], transactionId)
      );

      const prepareResults = await Promise.all(preparePromises);

      // Check if all voted YES
      const allPrepared = prepareResults.every(result => result.vote === 'YES');

      if (!allPrepared) {
        console.log(`❌ Not all participants prepared, ABORTING`);
        await this.abort(transactionId);
        return { success: false, reason: 'Prepare phase failed' };
      }

      console.log(`✅ All participants prepared`);
      context.preparedParticipants = this.participants;
      context.status = 'PREPARED';

      // PHASE 2: COMMIT
      console.log(`\n[PHASE 2] COMMIT`);
      console.log(`-${'='.repeat(50)}`);
    }
  }
}
```

```
const commitPromises = this.participants.map((participant, idx) =>
  this.commit(participant, transactionId)
);

await Promise.all(commitPromises);

console.log(✓ Transaction committed on all participants');
context.status = 'COMMITTED';

console.log('repeat(50));
console.log('Transaction completed successfully\n');

return { success: true };

} catch (error) {
  console.error(✗ Transaction failed:', error.message);

// Rollback
await this.abort(transactionId);
context.status = 'ABORTED';

return { success: false, error: error.message };
}

}

async prepare(participant, operation, transactionId) {
  console.log(` Sending PREPARE to ${participant.name}`);
  try {
    const canCommit = await participant.prepare(operation, transactionId);

    if (canCommit) {
      console.log(`${participant.name} voted YES`);
      return { vote: 'YES', participant };
    } else {
      console.log(`${participant.name} voted NO`);
      return { vote: 'NO', participant };
    }
  } catch (error) {
    console.log(`${participant.name} failed to respond`);
    return { vote: 'NO', participant };
  }
}

async commit(participant, transactionId) {
  console.log(` Sending COMMIT to ${participant.name}`);
}
```

```

await participant.commit(transactionId);
console.log(` ${participant.name} committed`);
}

async abort(transactionId) {
  console.log(`\n[ABORT] Rolling back transaction...`);

  for (const participant of this.participants) {
    try {
      await participant.abort(transactionId);
      console.log(` ${participant.name} aborted`);
    } catch (error) {
      console.error(` Failed to abort ${participant.name}`, error);
    }
  }
}

// Database participant
class DatabaseParticipant {
  constructor(name) {
    this.name = name;
    this.preparedTransactions = new Map();
    this.data = new Map();
  }

  async prepare(operation, transactionId) {
    // Check if can perform operation
    console.log(`[${this.name}] Preparing transaction ${transactionId}`);

    const { type, key, value } = operation;

    if (type === 'withdraw') {
      const current = this.data.get(key) || 0;
      if (current < value) {
        console.log(`[${this.name}] Insufficient funds`);
        return false; // Vote NO
      }
    }
  }

  // Lock resources and prepare
  this.preparedTransactions.set(transactionId, operation);
  console.log(`[${this.name}] Resources locked`);

  return true; // Vote YES
}

```

```
async commit(transactionId) {
  const operation = this.preparedTransactions.get(transactionId);

  if (!operation) {
    throw new Error('Transaction not prepared');
  }

  // Execute operation
  const { type, key, value } = operation;

  if (type === 'withdraw') {
    const current = this.data.get(key) || 0;
    this.data.set(key, current - value);
  } else if (type === 'deposit') {
    const current = this.data.get(key) || 0;
    this.data.set(key, current + value);
  }

  // Release locks
  this.preparedTransactions.delete(transactionId);

  console.log(`[${this.name}] Transaction committed`);
}

async abort(transactionId) {
  // Release locks without executing
  this.preparedTransactions.delete(transactionId);
  console.log(`[${this.name}] Transaction aborted`);

}

// Usage
const bankA = new DatabaseParticipant('Bank A');
const bankB = new DatabaseParticipant('Bank B');

// Set initial balances
bankA.data.set('account-123', 1000);
bankB.data.set('account-456', 500);

// Create coordinator
const coordinator = new TwoPhaseCommitCoordinator([bankA, bankB]);

// Transfer $100 from Bank A to Bank B
await coordinator.executeTransaction('txn-001', [
  { type: 'withdraw', key: 'account-123', value: 100 },
  { type: 'deposit', key: 'account-456', value: 100 }
]);
```

```

console.log('Bank A balance:', bankA.data.get('account-123')); // → 900
console.log('Bank B balance:', bankB.data.get('account-456')); // → 600

// Output shows complete 2PC flow:
// [PHASE 1] PREPARE
// Bank A voted YES
// Bank B voted YES
// [PHASE 2] COMMIT
// Bank A committed
// Bank B committed
// Transaction completed successfully

```

Two-Phase Commit Problems

Problems with 2PC:

1. BLOCKING PROTOCOL

If coordinator crashes during phase 2

→ Participants locked waiting

→ System blocked!

2. SINGLE POINT OF FAILURE

Coordinator is critical

If coordinator down, can't commit transactions

3. SLOW (Linear with participants)

3 participants: 2 round trips

10 participants: 2 round trips (but wait for slowest)

4. NOT PARTITION TOLERANT

If network partition, can't commit

Sacrifices availability

Real-World:

Most distributed databases DON'T use 2PC

Too slow, too fragile

Use eventual consistency instead

6. Consensus Algorithms

What is Consensus?

Problem: Multiple nodes must agree on a value.

Scenario: Leader Election

3 nodes must elect a leader:



Challenges:

- Nodes may fail
- Network may partition
- Messages may be lost/delayed
- Clocks are not synchronized

Goal: All nodes agree on same leader

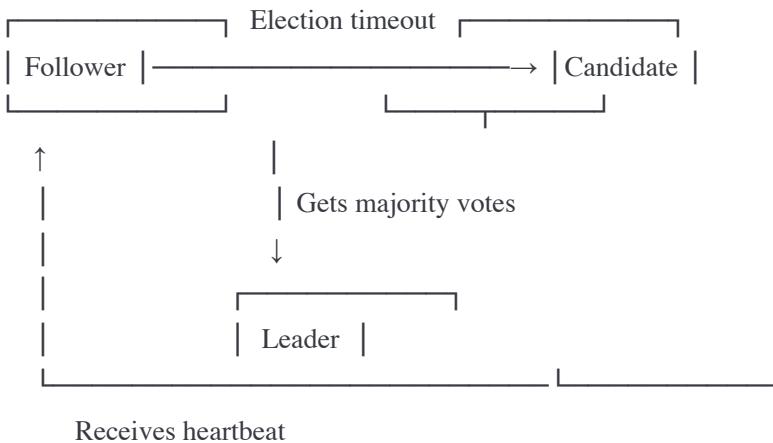
Even in presence of failures!

Raft Consensus Algorithm

Simpler alternative to Paxos. Used in etcd, Consul.

States:

Node can be in one of 3 states:



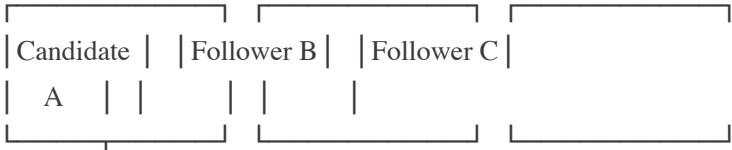
Leader Election:

Step-by-Step:

T=0: All nodes start as Followers



T=150ms: Node A's election timeout expires



Votes for self (1 vote)

Requests votes from others

"Vote for me?" —————> Node B

Node B: "Yes" ✓

"Vote for me?" —————> Node C

Node C: "Yes" ✓

T=170ms: Node A has 3 votes (majority!)



Sends heartbeats to maintain leadership

❤ —————> Node B

❤ —————> Node C

All nodes agree: Node A is leader!

Raft Implementation (Simplified)

javascript

```
class RaftNode {
  constructor(id, peers) {
    this.id = id;
    this.peers = peers;
    this.state = 'FOLLOWER'; // FOLLOWER, CANDIDATE, LEADER
    this.currentTerm = 0;
    this.votedFor = null;
    this.log = [];
    this.commitIndex = 0;

    // Timing
    this.electionTimeout = this.randomTimeout(150, 300);
    this.heartbeatInterval = 50;

    this.lastHeartbeat = Date.now();

    // Start election timer
    this.startElectionTimer();
  }

  randomTimeout(min, max) {
    return Math.floor(Math.random() * (max - min + 1)) + min;
  }

  startElectionTimer() {
    setInterval(() => {
      if (this.state !== 'LEADER') {
        const timeSinceHeartbeat = Date.now() - this.lastHeartbeat;

        if (timeSinceHeartbeat > this.electionTimeout) {
          console.log(`[${this.id}] Election timeout! Starting election...`);
          this.startElection();
        }
      }
    }, 50);
  }

  async startElection() {
    // Become candidate
    this.state = 'CANDIDATE';
    this.currentTerm++;
    this.votedFor = this.id;

    console.log(`[${this.id}] Became CANDIDATE for term ${this.currentTerm}`);

    let votes = 1; // Vote for self
```

```

const votesNeeded = Math.floor(this.peers.length / 2) + 1;

// Request votes from peers
const votePromises = this.peers.map(peer =>
  this.requestVote(peer)
);

const voteResults = await Promise.allSettled(votePromises);

voteResults.forEach(result => {
  if (result.status === 'fulfilled' && result.value) {
    votes++;
  }
});

console.log(`[${this.id}] Got ${votes} votes (need ${votesNeeded})`);

// Check if won election
if (votes >= votesNeeded && this.state === 'CANDIDATE') {
  this.becomeLeader();
} else {
  console.log(`[${this.id}] Lost election, back to FOLLOWER`);
  this.state = 'FOLLOWER';
}

async requestVote(peer) {
  try {
    const response = await peer.handleVoteRequest({
      term: this.currentTerm,
      candidateId: this.id,
      lastLogIndex: this.log.length - 1,
      lastLogTerm: this.log[this.log.length - 1]?.term || 0
    });

    if (response.voteGranted) {
      console.log(`[${this.id}] Got vote from ${peer.id}`);
      return true;
    }
  }

  return false;
}

} catch (error) {
  return false;
}
}

```

```

handleVoteRequest(request) {
    // Grant vote if:
    // 1. Haven't voted in this term yet
    // 2. Candidate's log is at least as up-to-date as ours

    if (request.term > this.currentTerm) {
        this.currentTerm = request.term;
        this.votedFor = null;
        this.state = 'FOLLOWER';
    }

    if (request.term < this.currentTerm) {
        return { voteGranted: false, term: this.currentTerm };
    }

    if (this.votedFor === null || this.votedFor === request.candidateId) {
        this.votedFor = request.candidateId;
        console.log(`[${this.id}] Voted for ${request.candidateId}`);
        return { voteGranted: true, term: this.currentTerm };
    }

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

becomeLeader() {
    console.log(`[${this.id}] 🎉 Became LEADER for term ${this.currentTerm}`);
    this.state = 'LEADER';

    // Send periodic heartbeats
    this.sendHeartbeats();
}

sendHeartbeats() {
    if (this.state !== 'LEADER') return;

    this.peers.forEach(peer => {
        peer.handleHeartbeat({
            term: this.currentTerm,
            leaderId: this.id
        });
    });

    setTimeout(() => this.sendHeartbeats(), this.heartbeatInterval);
}

handleHeartbeat(request) {
    if (request.term >= this.currentTerm) {

```

```
this.lastHeartbeat = Date.now();
this.currentTerm = request.term;

if (this.state !== 'FOLLOWER') {
  console.log(`[${this.id}] Stepping down to FOLLOWER`);
  this.state = 'FOLLOWER';
}

}

// Log replication (simplified)
async appendEntry(entry) {
  if (this.state !== 'LEADER') {
    throw new Error('Only leader can append entries');
  }

  console.log(`[${this.id}] Appending entry: ${JSON.stringify(entry)}`);

  // Add to leader's log
  this.log.push({
    term: this.currentTerm,
    ...entry
  });
}

// Replicate to followers
const replicationPromises = this.peers.map(peer =>
  peer.handleAppendEntries({
    term: this.currentTerm,
    leaderId: this.id,
    entries: [entry]
  })
);

const results = await Promise.allSettled(replicationPromises);

const successCount = results.filter(r => r.status === 'fulfilled').length + 1;
const majority = Math.floor(this.peers.length / 2) + 1;

if (successCount >= majority) {
  // Majority replicated, commit!
  this.commitIndex++;
  console.log(`[${this.id}] Entry committed (replicated to ${successCount} nodes}`);
  return { success: true };
} else {
  console.log(`[${this.id}] Failed to replicate to majority`);
  return { success: false };
}
```

```
}

handleAppendEntries(request) {
    // Accept entries from leader
    if (request.term >= this.currentTerm) {
        this.lastHeartbeat = Date.now();
        this.currentTerm = request.term;
        this.state = 'FOLLOWER';

        // Append entries
        request.entries.forEach(entry => {
            this.log.push({
                term: request.term,
                ...entry
            });
        });
    }

    return { success: true };
}

return { success: false };
}

// Create cluster
const nodeA = new RaftNode('A', []);
const nodeB = new RaftNode('B', []);
const nodeC = new RaftNode('C', []);

// Connect peers
nodeA.peers = [nodeB, nodeC];
nodeB.peers = [nodeA, nodeC];
nodeC.peers = [nodeA, nodeB];

// Simulation: Eventually one becomes leader
setTimeout(async () => {
    // Find leader
    const leader = [nodeA, nodeB, nodeC].find(node => node.state === 'LEADER');

    if (leader) {
        console.log(`\nLeader elected: ${leader.id}`);
    }

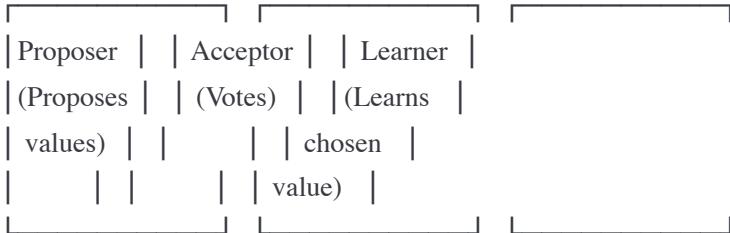
    // Leader can now accept writes
    await leader.appendEntry({
        command: 'SET',
        key: 'balance',
        value: 1000
    });
})
```

```
});  
}  
}, 2000);
```

Paxos Algorithm (Brief Overview)

Paxos: The original consensus algorithm (more complex than Raft).

Paxos Roles:



Phases:

Phase 1: Prepare

Proposer → Acceptors: "Prepare with number N"

Acceptors → Proposer: "Promise to not accept lower numbered proposals"

Phase 2: Accept

Proposer → Acceptors: "Accept value V with number N"

Acceptors → Proposer: "Accepted"

Phase 3: Learn

Acceptors → Learners: "Value V was chosen"

Properties:

- Can handle failures
- Can handle network issues
- Guarantees agreement
- Complex to implement correctly

Why Raft is Preferred:

Feature	Paxos	Raft
Understandability	Complex	Simpler
Leader Election	Separate	Integrated
Log Structure	Flexible	Sequential
Implementation	Hard	Easier
Correctness	Proven	Proven

Raft designed to be understandable

Used in: etcd, Consul, CockroachDB

Paxos used in: Google Chubby, Spanner

Real-World Applications

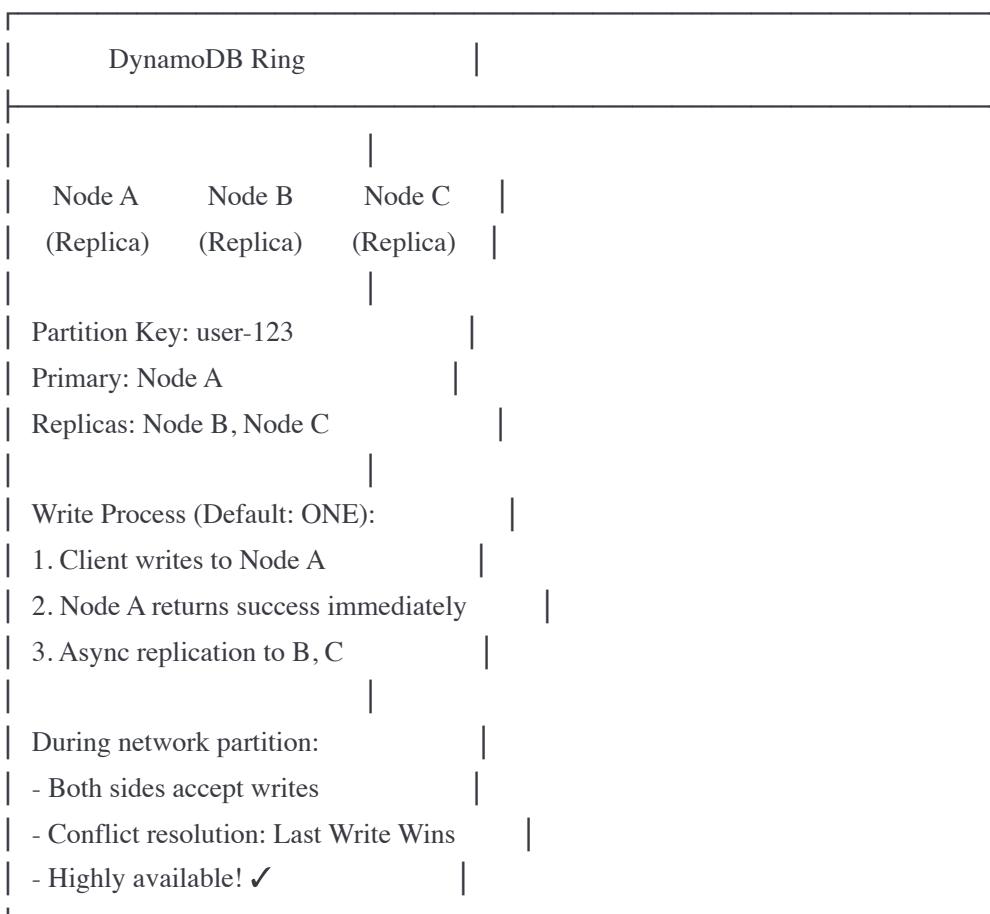
Example 1: DynamoDB (AP System)

Amazon DynamoDB:

Chose: Availability + Partition Tolerance

Sacrificed: Strong Consistency (uses eventual)

Architecture:



Configuration:

```
javascript

const AWS = require('aws-sdk');
const dynamodb = new AWS.DynamoDB.DocumentClient();

// Write with eventual consistency (fast)
await dynamodb.put({
  TableName: 'Users',
  Item: {
    userId: '123',
    name: 'John',
    balance: 1000
  }
  // Default: Eventually consistent replication
}).promise();

// Read with strong consistency (slower but guaranteed latest)
const result = await dynamodb.get({
  TableName: 'Users',
  Key: { userId: '123' },
  ConsistentRead: true // Strong consistency
}).promise();

// Read with eventual consistency (fast but might be stale)
const result2 = await dynamodb.get({
  TableName: 'Users',
  Key: { userId: '123' },
  ConsistentRead: false // Default: eventual consistency
}).promise();
```

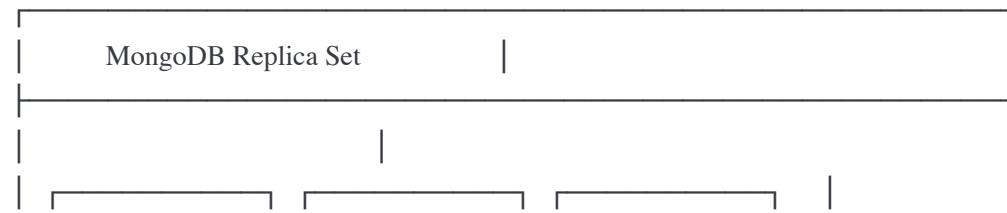
Example 2: MongoDB (CP System)

MongoDB:

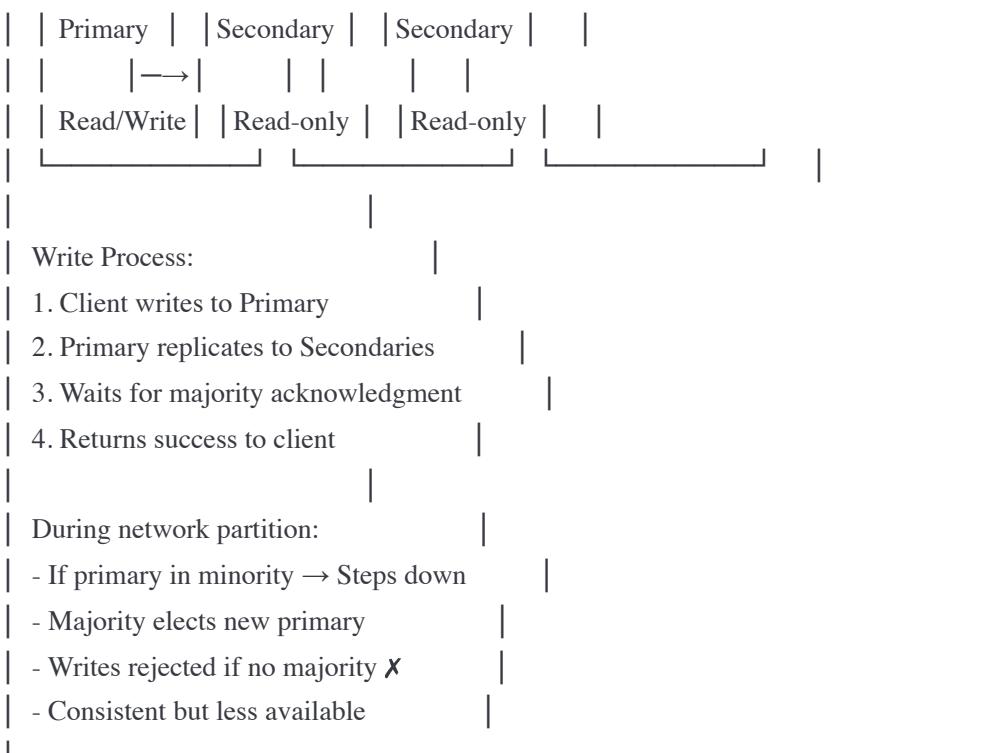
Chose: Consistency + Partition Tolerance

Sacrificed: Availability (rejects writes if can't reach majority)

Architecture:



```
graph LR; A[MongoDB Replica Set] --- B[ ]; B --- C[ ]; B --- D[ ]; B --- E[ ]
```



Configuration:

javascript

```

const { MongoClient } = require('mongodb');

const client = new MongoClient('mongodb://mongo1,mongo2,mongo3/?replicaSet=rs0');

// Write with majority acknowledgment (strong consistency)
await db.collection('accounts').updateOne(
  { userId: 123 },
  { $set: { balance: 1000 } },
  {
    writeConcern: {
      w: 'majority', // Wait for majority
      j: true, // Wait for journal
      wtimeout: 5000 // Timeout after 5s
    }
  }
);

// Read from primary (strong consistency)
const result = await db.collection('accounts').findOne(
  { userId: 123 },
  {
    readPreference: 'primary' // Always read from primary
  }
);

// Read from secondary (eventual consistency, faster)
const result2 = await db.collection('accounts').findOne(
  { userId: 123 },
  {
    readPreference: 'secondary' // May be slightly stale
  }
);

// Trade-off:
// Majority write: Slower but consistent
// Secondary read: Faster but may be stale

```

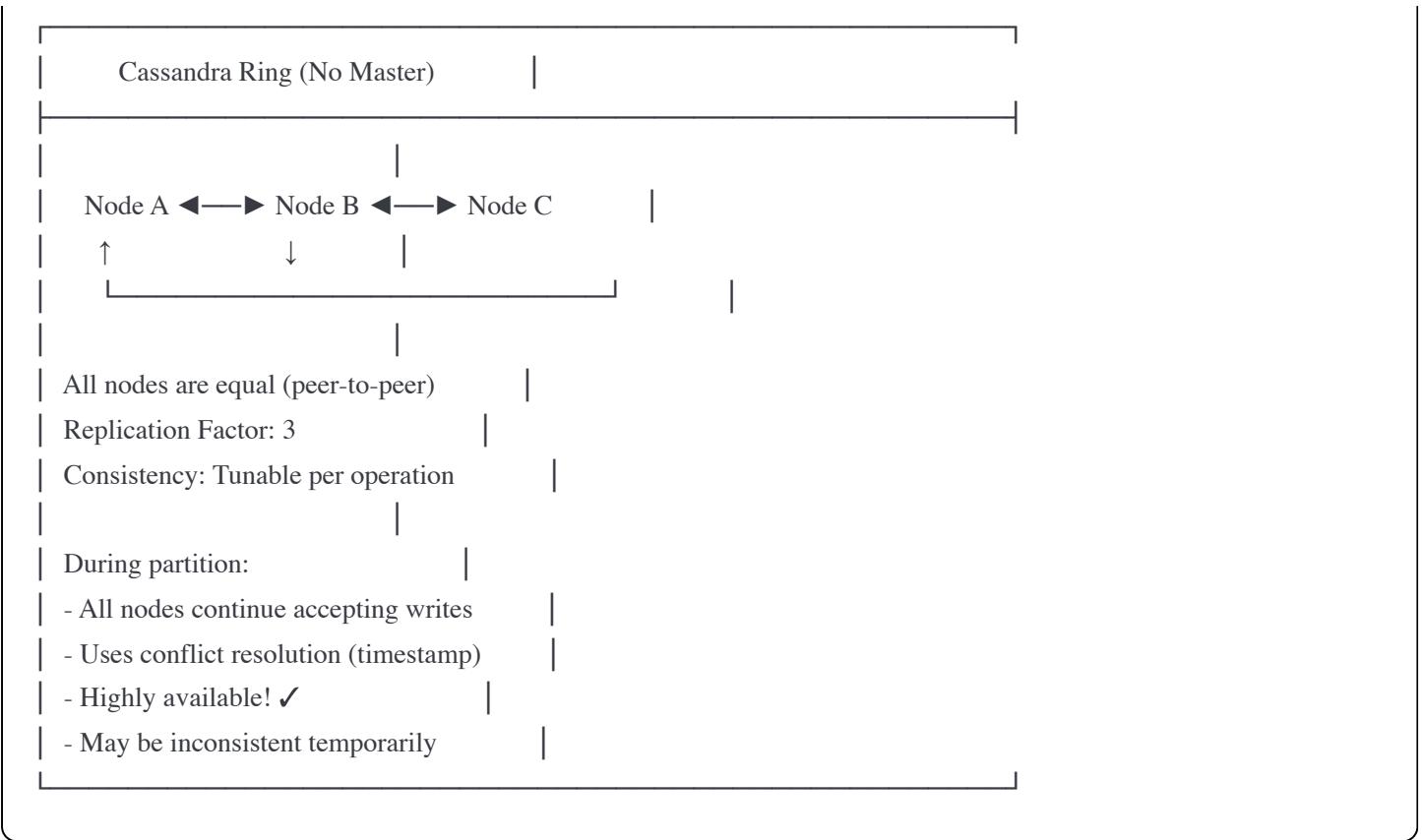
Example 3: Cassandra (AP System)

Cassandra:

Chose: Availability + Partition Tolerance

Sacrificed: Strong Consistency (tunable though!)

Architecture:



Key Takeaways

1. CAP Theorem:

- Can't have all three: C, A, P
- Network partitions happen (must handle P)
- Choose between C and A
- CP: Banks, inventory
- AP: Social media, caching

2. ACID vs BASE:

- ACID: Strong guarantees, limited scale
- BASE: Eventual consistency, high scale
- Trade-off: Correctness vs Performance

3. Eventual Consistency:

- Fast writes (don't wait)
- May read stale data
- Eventually converges
- Requires conflict resolution

4. Strong Consistency:

- Wait for replication
- Always read latest
- Slower but correct
- Lower availability

5. Distributed Transactions:

- 2PC: Atomic but blocking
- Avoid if possible (use Saga instead)
- Slow and fragile

6. Consensus Algorithms:

- Raft: Easier to understand
- Paxos: Original, complex
- Needed for leader election, replication
- Used by: etcd, Consul, Zookeeper

Practice Problems

1. Your system needs 99.99% availability but also strong consistency. Is this possible? What trade-offs?
2. Design a system for bank transfers between different banks. Which consistency model?
3. Design a social media feed. Which consistency model and why?
4. Explain why 2PC is rarely used in practice and what alternatives exist.

Next Chapter Preview

Chapter 17: Data Processing at Scale

- Batch vs Stream processing
- MapReduce
- Apache Spark
- Lambda and Kappa architectures

Ready to continue?