Chapter 12: High Availability

Introduction: What is High Availability?

High Availability (HA): Ensuring a system remains operational and accessible even when components fail.

```
Low Availability System:
┌─────────────────────────────────────┐
│ Server crashes → Site down          │
│ Downtime: 2 hours per week          │
│ Users: Cannot access service        │
│ Revenue: Lost                       │
│ Reputation: Damaged                 │
└─────────────────────────────────────┘

High Availability System:
┌─────────────────────────────────────┐
│ Server crashes → Backup takes over  │
│ Downtime: 30 seconds                │
│ Users: Brief interruption           │
│ Revenue: Protected                  │
│ Reputation: Maintained              │
└─────────────────────────────────────┘
```

---

1. Understanding the Nine Nines

Availability Percentage to Downtime

Formula:

```
Availability = (Total Time - Downtime) / Total Time × 100%
Uptime = Total Time - Downtime
```

The Nine Nines Table:

Availability	Downtime/Year	Downtime/Mo	Downtime/Week	Downtime/Day
90% (one nine)	36.5 days	3 days	16.8 hours	2.4 hours
99% (two nines)	3.65 days	7.2 hours	1.68 hours	14.4 min
99.9% (three nines	8.76 hours	43.2 min	10.1 min	1.44 min
99.99% (four nines	52.6 min	4.32 min	1.01 min	8.64 sec
99.999% (five nines	5.26 min	26 sec	6.05 sec	0.86 sec
99.9999% (six nines	31.5 sec	2.59 sec	0.605 sec	0.086 sec

Key Insight: Each additional nine gets exponentially harder!
99% → 99.9% is much easier than 99.99% → 99.999%

Real-World Availability Requirements

Service Type	Target SLA	Acceptable?
Personal blog	90%	Yes (hobby)
Small business site	95%	Acceptable
E-commerce	99.9%	Minimum
Banking	99.99%	Required
Payment processing	99.99%	Required
Emergency services	99.999%	Critical
Life support systems	99.9999%	Essential

Cost of Downtime:

E-commerce Site Example:
- Revenue: $10 million/year
- Revenue per hour: $10M / 8760 hours = $1,141/hour

Availability	Downtime/Year	Lost Revenue
99%	3.65 days	$100,000/year
99.9%	8.76 hours	$10,000/year
99.99%	52.6 min	$1,000/year
99.999%	5.26 min	$100/year

Going from 99% → 99.9% saves $90,000/year!

Calculating System Availability

Serial Components (All Must Work)

Formula: Total Availability = $A_1 \times A_2 \times A_3 \times \ldots \times A_n$

Example: Web Application Stack

```
┌─────────────────┐
│ Load Balancer │ 99.99%
└────────┬────────┘
     ↓
┌─────────────────┐
│ Web Server  │ 99.95%
└────────┬────────┘
     ↓
┌─────────────────┐
│ Database   │ 99.9%
└─────────────────┘
```

Total = 99.99% × 99.95% × 99.9%
  = 0.9999 × 0.9995 × 0.999
  = 0.9984
  = 99.84%

Downtime: 14 hours/year

Key Point: Weakest link drags down overall availability!

Parallel Components (Redundancy)

```
Formula: Total Availability = 1 - ((1 - A₁) × (1 - A₂) × ... × (1 - Aₙ))

Example: Two Load Balancers (Failover)
┌──────────────────┐
│Load Balancer │ 99.9%
│      1       │
└──────────────────┘

┌──────────────────┐
│Load Balancer │ 99.9%
│      2       │
└──────────────────┘

Total = 1 - ((1 - 0.999) × (1 - 0.999))
      = 1 - (0.001 × 0.001)
      = 1 - 0.000001
      = 0.999999
      = 99.9999%

Downtime: 31 seconds/year

Key Point: Redundancy dramatically improves availability!
```

Code to Calculate Availability:

```python
```

```python
def calculate_serial_availability(components):
    """Calculate availability when all components must work"""
    total = 1.0
    for component in components:
        total *= component
    return total

def calculate_parallel_availability(components):
    """Calculate availability with redundant components"""
    total_failure = 1.0
    for component in components:
        total_failure *= (1 - component)
    return 1 - total_failure

def downtime_per_year(availability):
    """Calculate downtime from availability percentage"""
    year_minutes = 365 * 24 * 60
    downtime_minutes = year_minutes * (1 - availability)

    if downtime_minutes > 60:
        hours = downtime_minutes / 60
        return f"{hours:.2f} hours"
    else:
        return f"{downtime_minutes:.2f} minutes"

# Example: Web stack
print("Serial Components (all must work):")
web_stack = [0.9999, 0.9995, 0.999]  # LB, Web, DB
total = calculate_serial_availability(web_stack)
print(f"Availability: {total*100:.2f}%")
print(f"Downtime: {downtime_per_year(total)}")

# Output:
# Availability: 99.84%
# Downtime: 14.01 hours

print("\nParallel Components (redundancy):")
load_balancers = [0.999, 0.999]
total = calculate_parallel_availability(load_balancers)
print(f"Availability: {total*100:.4f}%")
print(f"Downtime: {downtime_per_year(total)}")

# Output:
# Availability: 99.9999%
# Downtime: 0.53 minutes
```

```
# Complex system calculation
print("\nComplete System:")
# 2 redundant load balancers
lb_availability = calculate_parallel_availability([0.999, 0.999])
print(f"Load Balancer layer: {lb_availability*100:.4f}%")

# 3 redundant web servers
web_availability = calculate_parallel_availability([0.999, 0.999, 0.999])
print(f"Web server layer: {web_availability*100:.6f}%")

# 1 database with replication
db_availability = 0.9995
print(f"Database layer: {db_availability*100:.2f}%")

# Total (serial combination of layers)
total = calculate_serial_availability([
    lb_availability,
    web_availability,
    db_availability
])
print(f"\nTotal System Availability: {total*100:.4f}%")
print(f"Downtime: {downtime_per_year(total)}")

# Output:
# Load Balancer layer: 99.9999%
# Web server layer: 99.999999%
# Database layer: 99.95%
# Total System Availability: 99.9499%
# Downtime: 4.38 hours
```

---

2. Single Points of Failure (SPOF)

What is a Single Point of Failure?

Definition: A component whose failure causes entire system to fail.

```
System with SPOF:

┌─────────────────────────────────────┐
│  Users                              │
│    ↓                                │
│  ┌──────────────────┐               │
│  │ Load Balancer    │ ← SPOF!       │
│  │   (single)       │               │
│  └────────┬─────────┘               │
│           │                         │
│     ┌─────┴─────┬──────┐            │
│     ↓           ↓      ↓            │
│  ┌─────┐   ┌─────┐   ┌─────┐        │
│  │ Web │   │ Web │   │ Web │        │
│  │  1  │   │  2  │   │  3  │        │
│  └─────┘   └─────┘   └─────┘        │
└─────────────────────────────────────┘

If load balancer fails → Entire site down!
Even though web servers are healthy.
```

Identifying SPOFs

Checklist:

```
1. SINGLE SERVER/SERVICE
  ❌ One database server
  ❌ One load balancer
  ❌ One cache server
  ❌ One message queue
  ✓ Multiple instances of each

2. SINGLE NETWORK PATH
  ❌ One internet connection
  ❌ One network switch
  ❌ One data center
  ✓ Multiple paths, multiple DCs

3. SINGLE DEPENDENCY
  ❌ One payment gateway (what if it's down?)
  ❌ One authentication service
  ✓ Multiple providers or fallback mechanism

4. SINGLE PERSON/TEAM
  ❌ Only one person knows how to deploy
  ❌ Only one team can fix production
  ✓ Documentation, cross-training, automation
```

Finding SPOFs - Systematic Approach:

```python
```

```python
class SPOFAnalyzer:
    """Tool to identify single points of failure"""

    def __init__(self):
        self.components = {}

    def add_component(self, name, instances, critical=True):
        """Register a component"""
        self.components[name] = {
            'instances': instances,
            'critical': critical
        }

    def analyze(self):
        """Find single points of failure"""
        spofs = []

        for name, info in self.components.items():
            if info['critical'] and info['instances'] == 1:
                spofs.append(name)

        return spofs

    def calculate_risk(self):
        """Calculate system risk"""
        print("="*50)
        print("SINGLE POINT OF FAILURE ANALYSIS")
        print("="*50)

        spofs = self.analyze()

        if spofs:
            print(f"\n⚠️  Found {len(spofs)} single points of failure:")
            for spof in spofs:
                print(f"  - {spof} (1 instance)")
            print("\n💡  Recommendation: Add redundancy to these components")
        else:
            print("\n✓ No single points of failure detected")

        # Calculate worst-case availability
        for name, info in self.components.items():
            if info['instances'] == 1:
                print(f"\nIf {name} fails → TOTAL SYSTEM FAILURE")

# Usage
analyzer = SPOFAnalyzer()
```

```python
# Define system components
analyzer.add_component('Load Balancer', instances=1, critical=True)  # SPOF!
analyzer.add_component('Web Server', instances=3, critical=True)
analyzer.add_component('Database', instances=1, critical=True)       # SPOF!
analyzer.add_component('Cache', instances=2, critical=False)

analyzer.calculate_risk()

# Output:
# ⚠️  Found 2 single points of failure:
#   - Load Balancer (1 instance)
#   - Database (1 instance)
#
# If Load Balancer fails → TOTAL SYSTEM FAILURE
# If Database fails → TOTAL SYSTEM FAILURE
```

Eliminating SPOFs - Before and After

```
BEFORE (Multiple SPOFs):

┌─────────────────────────────────────────┐
│         Users                    │
│           ↓                    │
│     ┌──────────────┐             │
│     │  DNS    │ ← SPOF 1    │
│     └──────┬───────┘             │
│           ↓                    │
│     ┌──────────────┐             │
│     │  LB    │ ← SPOF 2    │
│     └──────┬───────┘             │
│           │            │
│     ┌──────┴────────┬──────┐         │
│    ↓       ↓  ↓     │
│   ┌────┐   ┌────┐ ┌────┐     │
│   │Web1│   │Web2│ │Web3│     │
│   └─┬──┘   └─┬──┘ └─┬──┘     │
│    └─────────────┼──────┘       │
│          ↓              │
│      ┌──────────────┐          │
│      │ Database  │ ← SPOF 3 │
│      └──────────────┘          │
└─────────────────────────────────────────┘

AFTER (No SPOFs):

┌─────────────────────────────────────────┐
```

```
│         Users                    │
│           ↓                    │
│   ┌───────────────────────────┐           │
│   │ DNS (Route53 / NS1)  │ ✓ Managed │
│   │   Multi-provider     │        │
│   └──────────────┬────────────┘           │
│         │                   │
│    ┌─────────┴─────────┐           │
│    ↓           ↓            │
│   ┌─────┐        ┌─────┐          │
│   │ LB1 │       │LB2 │ ✓ Redundant │
│   └──┬──┘        └──┬──┘          │
│    └────────┬────────┘            │
│        │                 │
│    ┌──────┴──────┬────────┐         │
│    ↓       ↓    ↓        │
│   ┌─────┐     ┌─────┐ ┌─────┐       │
│   │Web1 │     │Web2 │ │Web3 │ ✓ Multiple │
│   └──┬──┘     └──┬──┘ └──┬──┘       │
│    └────────────┼───────┘          │
│           ↓              │
│      ┌─────────────────┐          │
│      │DB Primary    │      │
│      └───────┬─────────┘          │
│          │ Replication     │
│      ┌───────┴─────────┐          │
│       ↓        ↓        │
│    ┌────────────┐  ┌────────────┐   │
│    │DB Replica  │DB Replica │ ✓ Replicated │
│    │   1    │  │   2    │   │
│    └────────────┘  └────────────┘   │
└───────────────────────────────────────────┘
```

No single points of failure!

---

3. Redundancy and Replication

Types of Redundancy

1. Active Redundancy (Hot Standby)

```
Both systems running simultaneously:

┌─────────────┐   ┌─────────────┐
│ Primary     │   │ Standby     │
│ Server      │   │ Server      │
│             │   │             │
│ Active ✓    │   │ Active ✓    │
│ Serving     │◄─────►│ Serving     │
│ Traffic     │ Sync │ Traffic     │
└─────────────┘   └─────────────┘

Benefits:
- Instant failover (both already running)
- No cold start time
- Load balanced
- Zero data loss

Cost:
- 2x resources
- Both servers doing work
```

Implementation:

```python
```

```python
import time
import threading
import requests

class ActiveActiveServers:
    def __init__(self):
        self.servers = [
            {'url': 'http://server1:8000', 'healthy': True},
            {'url': 'http://server2:8000', 'healthy': True}
        ]
        self.current_index = 0

        # Start health check thread
        self.monitoring = True
        self.health_thread = threading.Thread(target=self._health_check_loop)
        self.health_thread.daemon = True
        self.health_thread.start()

    def _health_check_loop(self):
        """Continuously check server health"""
        while self.monitoring:
            for server in self.servers:
                try:
                    response = requests.get(
                        f"{server['url']}/health",
                        timeout=2
                    )
                    server['healthy'] = (response.status_code == 200)
                except:
                    server['healthy'] = False

            time.sleep(5)  # Check every 5 seconds

    def get_server(self):
        """Get next healthy server (round-robin)"""
        attempts = 0
        max_attempts = len(self.servers)

        while attempts < max_attempts:
            server = self.servers[self.current_index]
            self.current_index = (self.current_index + 1) % len(self.servers)

            if server['healthy']:
                return server

            attempts += 1
```

```python
        raise Exception("No healthy servers available")

    def handle_request(self, request):
        """Route request to healthy server"""
        server = self.get_server()

        try:
            response = requests.post(
                f"{server['url']}/api/process",
                json=request,
                timeout=5
            )
            return response.json()

        except Exception as e:
            # Mark server as unhealthy
            server['healthy'] = False

            # Retry with another server
            print(f"Server {server['url']} failed, retrying...")
            return self.handle_request(request)

# Usage
servers = ActiveActiveServers()

# Both servers handling requests
for i in range(10):
    result = servers.handle_request({'task': f'task-{i}'})
    print(f"Task {i} processed by server")

# If server1 goes down, server2 takes all traffic automatically!
```

---

2. Passive Redundancy (Warm Standby)

```
Primary active, standby ready but not serving:

┌─────────────┐   ┌─────────────┐
│  Primary    │   │  Standby    │
│  Server     │   │  Server     │
│             │   │             │
│ Active ✓    │   │ Idle ⏸      │
│ Serving     │──────→│ Ready       │
│ Traffic     │ Sync │ Not serving │
└─────────────┘   └─────────────┘

On primary failure:
┌─────────────┐   ┌─────────────┐
│  Primary    │   │  Standby    │
│  Server     │   │  Server     │
│             │   │             │
│ Failed ✗    │   │ Active ✓    │
│             │   │ Now serving │
│             │   │ Traffic     │
└─────────────┘   └─────────────┘

Benefits:
- Faster failover than cold standby
- Lower cost than active-active
- Standby is ready

Drawbacks:
- Some failover time (seconds)
- Resources idle (waste)
```

---

3. Cold Standby (Backup)

```
Standby is off, must boot on failure:

┌─────────────┐     ┌─────────────┐
│  Primary    │     │  Standby    │
│  Server     │     │  Server     │
│             │     │             │
│ Active ✓    │     │ Offline ⏸   │
│ Serving     │────→│ Backup ready│
│ Traffic     │Sync │ Not running │
└─────────────┘     └─────────────┘

On primary failure:
1. Detect failure (30 sec)
2. Start standby (2 min)
3. Restore from backup (5 min)
4. Begin serving traffic (7.5 min)

Benefits:
- Lowest cost
- Still have backup

Drawbacks:
- Long failover time (minutes to hours)
- Potential data loss
- Complex restoration
```

Replication Strategies

Strategy	Failover	Cost	Data Loss
Active-Active (Hot)	Instant (0 seconds)	Highest (2x)	None
Active-Passive (Warm)	Fast (1-30 sec)	Medium (1.5x)	Minimal
Backup (Cold)	Slow (minutes)	Lowest (1.1x)	Possible

---

4. Failover Mechanisms

Automatic Failover

Health Check Based Failover:

```javascript
```

```
class AutomaticFailover {
  constructor(primary, backup) {
    this.primary = primary;
    this.backup = backup;
    this.currentActive = this.primary;
    this.failoverInProgress = false;

    // Start monitoring
    this.startHealthChecks();
  }

  startHealthChecks() {
    setInterval(() => {
      this.checkHealth();
    }, 5000);  // Check every 5 seconds
  }

  async checkHealth() {
    // Check primary
    const primaryHealthy = await this.isHealthy(this.primary);
    const backupHealthy = await this.isHealthy(this.backup);

    console.log(`Health: Primary=${primaryHealthy}, Backup=${backupHealthy}`);

    // Failover logic
    if (this.currentActive === this.primary && !primaryHealthy) {
      if (backupHealthy) {
        console.log('🚨 PRIMARY FAILED - Initiating failover to backup');
        await this.failoverToBackup();
      } else {
        console.log('🚨 BOTH SERVERS DOWN - CRITICAL!');
        this.alertOps('Both primary and backup are down!');
      }
    }

    // Failback when primary recovers
    if (this.currentActive === this.backup && primaryHealthy) {
      console.log('Primary recovered - Failing back to primary');
      await this.failbackToPrimary();
    }
  }

  async isHealthy(server) {
    """Check if server is responding"""
    try {
      const response = await fetch(`${server.url}/health`, {
```

```
      timeout: 2000
    });
    return response.status === 200;
  } catch (error) {
    return false;
  }
}

async failoverToBackup() {
  if (this.failoverInProgress) {
    return; // Already failing over
  }

  this.failoverInProgress = true;

  try {
    console.log('[1/4] Starting failover process...');

    // Step 1: Verify backup is ready
    console.log('[2/4] Verifying backup health...');
    if (!await this.isHealthy(this.backup)) {
      throw new Error('Backup is not healthy!');
    }

    // Step 2: Promote backup to active
    console.log('[3/4] Promoting backup to active...');
    await this.promoteToActive(this.backup);

    // Step 3: Update routing
    console.log('[4/4] Updating traffic routing...');
    this.currentActive = this.backup;

    // Step 4: Update DNS (if needed)
    await this.updateDNS(this.backup.ip);

    console.log('✅ Failover completed successfully');
    console.log(`Active server: ${this.currentActive.url}`);

    // Alert ops team
    this.alertOps(`Failover completed: Now serving from ${this.backup.url}`);

  } catch (error) {
    console.error('❌ Failover failed:', error);
    this.alertOps(`CRITICAL: Failover failed - ${error.message}`);
  } finally {
    this.failoverInProgress = false;
  }
```

```
  }

  async failbackToPrimary() {
    console.log('Starting failback to primary...');

    // Ensure primary is healthy for sustained period (30 seconds)
    for (let i = 0; i < 6; i++) {
      if (!await this.isHealthy(this.primary)) {
        console.log('Primary not stable, aborting failback');
        return;
      }
      await new Promise(resolve => setTimeout(resolve, 5000));
    }

    // Failback
    console.log('Primary is stable, failing back...');
    this.currentActive = this.primary;
    await this.updateDNS(this.primary.ip);

    console.log('✅ Failback completed');
  }

  async promoteToActive(server) {
    // For database: promote replica to primary
    // For application: ensure it's in read-write mode
    console.log(`Promoting ${server.url} to active status`);
  }

  async updateDNS(newIP) {
    // Update DNS to point to new server
    console.log(`Updating DNS to ${newIP}`);
    // In production: Use Route53, CloudFlare API, etc.
  }

  alertOps(message) {
    console.log(`📧 ALERT: ${message}`);
    // Send email, Slack, PagerDuty, etc.
  }
}

// Usage
const failover = new AutomaticFailover(
  { url: 'http://primary-server:8000', ip: '10.0.0.1' },
  { url: 'http://backup-server:8000', ip: '10.0.0.2' }
);
```

```
// System automatically monitors and fails over when needed!
```

Database Failover Example (PostgreSQL)

```
python
```

```python
import psycopg2
import time

class DatabaseFailover:
    def __init__(self):
        self.primary = {
            'host': 'primary-db.example.com',
            'port': 5432,
            'database': 'myapp',
            'user': 'app',
            'password': 'password'
        }

        self.replicas = [
            {
                'host': 'replica1-db.example.com',
                'port': 5432,
                'database': 'myapp',
                'user': 'app',
                'password': 'password'
            },
            {
                'host': 'replica2-db.example.com',
                'port': 5432,
                'database': 'myapp',
                'user': 'app',
                'password': 'password'
            }
        ]

        self.current_primary = self.primary
        self.connection = None

    def connect(self):
        """Connect to database with automatic failover"""
        try:
            self.connection = psycopg2.connect(**self.current_primary)
            return self.connection
        except Exception as e:
            print(f"Failed to connect to {self.current_primary['host']}: {e}")
            return self.failover()

    def failover(self):
        """Failover to replica"""
        print("🚨 Initiating database failover...")
```

```
        for replica in self.replicas:
            try:
                print(f"Attempting to promote replica: {replica['host']}")

                # Try to connect to replica
                conn = psycopg2.connect(**replica)

                # Check if replica is up-to-date
                cursor = conn.cursor()
                cursor.execute("SELECT pg_is_in_recovery();")
                is_replica = cursor.fetchone()[0]

                if is_replica:
                    # Promote replica to primary
                    print(f"Promoting {replica['host']} to primary...")
                    cursor.execute("SELECT pg_promote();")

                    # Wait for promotion
                    time.sleep(2)

                    # Verify promotion
                    cursor.execute("SELECT pg_is_in_recovery();")
                    still_replica = cursor.fetchone()[0]

                    if not still_replica:
                        print(f"✅ Successfully promoted {replica['host']}")
                        self.current_primary = replica
                        self.connection = conn

                        # Alert ops
                        self.alert_ops(f"Database failover: {replica['host']} is now primary")

                        return conn
                else:
                    # Already primary (shouldn't happen)
                    self.connection = conn
                    return conn

            except Exception as e:
                print(f"Failed to promote {replica['host']}: {e}")
                continue

        raise Exception("All database servers unavailable!")

    def execute(self, query, params=None):
        """Execute query with automatic retry on failure"""
        max_retries = 3
```

```python
        for attempt in range(max_retries):
            try:
                if not self.connection:
                    self.connect()

                cursor = self.connection.cursor()
                cursor.execute(query, params or ())
                return cursor.fetchall()

            except Exception as e:
                print(f"Query failed (attempt {attempt + 1}/{max_retries}): {e}")

                if attempt < max_retries - 1:
                    # Try to failover
                    try:
                        self.failover()
                    except:
                        time.sleep(1)  # Wait before retry
                else:
                    raise

        raise Exception("Query failed after all retries")

    def alert_ops(self, message):
        print(f"📧 ALERT: {message}")
        # Send to PagerDuty, email, Slack, etc.

# Usage
db = DatabaseFailover()

# Normal operation
result = db.execute("SELECT * FROM users WHERE id = %s", (123,))

# If primary fails, automatically fails over to replica!
# Application continues working with minimal disruption
```

---

5. Active-Active vs Active-Passive Setups

Active-Passive (Master-Standby)

Architecture:

```
┌─────────────────────────────────────────────────┐
│      Active-Passive Setup          │
├─────────────────────────────────────────────────┤
│                            │
│  ┌─────────────────┐   ┌─────────────────┐  │
│  │  PRIMARY    │──────→│  STANDBY   │  │
│  │  (Active)   │    │  (Passive)  │  │
│  │          │    │          │  │
│  │ ✓ Serving   │    │ ⏸ Monitoring │  │
│  │ ✓ Read/Write │    │ ⏸ Ready     │  │
│  └─────────────────┘   └─────────────────┘  │
│      ↑                     │
│      │ All traffic              │
│   ┌──────┴──────┐                  │
│   │  Users  │                 │
│   └─────────────┘                  │
└─────────────────────────────────────────────────┘

Characteristics:
- One server active, one waiting
- Standby monitors primary
- On failure, standby becomes active
- Failover time: 10-60 seconds
```

Implementation with Heartbeat:

```python
```

```python
import socket
import time
import threading

class ActivePassiveHA:
    def __init__(self, is_primary=True, peer_ip=None):
        self.is_primary = is_primary
        self.peer_ip = peer_ip
        self.is_active = is_primary  # Primary starts active
        self.last_heartbeat = time.time()
        self.heartbeat_timeout = 10  # seconds

        if is_primary:
            # Primary sends heartbeats
            threading.Thread(target=self._send_heartbeats, daemon=True).start()
        else:
            # Standby receives heartbeats
            threading.Thread(target=self._receive_heartbeats, daemon=True).start()
            threading.Thread(target=self._monitor_primary, daemon=True).start()

    def _send_heartbeats(self):
        """Primary sends heartbeat to standby"""
        sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

        while True:
            try:
                message = f"HEARTBEAT:{time.time()}"
                sock.sendto(message.encode(), (self.peer_ip, 8888))
                print("💓  Heartbeat sent to standby")
            except Exception as e:
                print(f"Error sending heartbeat: {e}")

            time.sleep(3)  # Send every 3 seconds

    def _receive_heartbeats(self):
        """Standby receives heartbeat from primary"""
        sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
        sock.bind(('0.0.0.0', 8888))
        sock.settimeout(1)

        while True:
            try:
                data, addr = sock.recvfrom(1024)
                message = data.decode()

                if message.startswith('HEARTBEAT:'):
```

```
                self.last_heartbeat = time.time()
                print("💓 Heartbeat received from primary")

            except socket.timeout:
                continue
            except Exception as e:
                print(f"Error receiving heartbeat: {e}")

    def _monitor_primary(self):
        """Standby monitors primary health"""
        while True:
            time.sleep(1)

            if not self.is_active:
                time_since_heartbeat = time.time() - self.last_heartbeat

                if time_since_heartbeat > self.heartbeat_timeout:
                    print(f"🚨 No heartbeat for {time_since_heartbeat:.0f} seconds")
                    print("🚨 PRIMARY FAILED - Taking over!")
                    self.takeover()

    def takeover(self):
        """Standby becomes active"""
        if self.is_active:
            return  # Already active

        print("\n" + "="*50)
        print("     FAILOVER IN PROGRESS")
        print("="*50)

        # Step 1: Verify primary is really down
        print("[1/5] Verifying primary is down...")
        for i in range(3):
            if self._check_primary_health():
                print("Primary is up! Aborting failover.")
                return
            time.sleep(1)

        # Step 2: Become active
        print("[2/5] Promoting standby to active...")
        self.is_active = True

        # Step 3: Take over virtual IP
        print("[3/5] Taking over virtual IP...")
        self._claim_virtual_ip()

        # Step 4: Promote database (if applicable)
```

```
        print("[4/5] Promoting database to read-write...")
        self._promote_database()

        # Step 5: Start accepting traffic
        print("[5/5] Now accepting traffic...")

        print("="*50)
        print("    FAILOVER COMPLETED")
        print("="*50)

        # Alert
        self.alert_ops("Failover completed - Standby is now active")

    def _check_primary_health(self):
        """Final check before taking over"""
        try:
            response = requests.get(f"http://{self.peer_ip}:8000/health", timeout=2)
            return response.status_code == 200
        except:
            return False

    def _claim_virtual_ip(self):
        """Take over virtual IP address"""
        # In production: Use VRRP (Virtual Router Redundancy Protocol)
        # Or cloud provider's floating IP
        print("Virtual IP claimed: 10.0.0.100 now points to this server")

    def _promote_database(self):
        """Promote read replica to primary"""
        # Execute: pg_ctl promote
        print("Database promoted to read-write mode")

    def alert_ops(self, message):
        print(f"\n📧 ALERT TO OPS TEAM: {message}\n")

# Run primary server
# primary = ActivePassiveHA(is_primary=True, peer_ip='10.0.0.2')

# Run standby server
# standby = ActivePassiveHA(is_primary=False, peer_ip='10.0.0.1')

# When primary fails, standby automatically takes over!
```

Manual Failover

When to use manual failover:

- Planned maintenance
- Testing failover procedures
- Complex systems where automation risky

```python
```

```python
class ManualFailoverController:
    def __init__(self):
        self.primary = {'host': 'primary.db.com', 'status': 'active'}
        self.standby = {'host': 'standby.db.com', 'status': 'standby'}

    def initiate_planned_failover(self):
        """
        Planned failover with zero data loss
        Used for maintenance
        """
        print("\n" + "="*60)
        print("        PLANNED FAILOVER PROCEDURE")
        print("="*60)

        # Step 1: Verify readiness
        print("\n[Step 1/8] Pre-failover checks...")
        if not self._verify_standby_caught_up():
            print("❌ Standby not caught up with primary!")
            return False
        print("✓ Standby is caught up")

        # Step 2: Set primary to read-only
        print("\n[Step 2/8] Setting primary to read-only mode...")
        self._set_read_only(self.primary)
        print("✓ Primary is now read-only (no new writes)")

        # Step 3: Wait for final replication
        print("\n[Step 3/8] Waiting for final replication...")
        time.sleep(5)  # Ensure all writes replicated

        if not self._verify_standby_caught_up():
            print("❌ Standby still not caught up!")
            self._set_read_write(self.primary)  # Rollback
            return False
        print("✓ Standby has all data")

        # Step 4: Verify standby health
        print("\n[Step 4/8] Final health check on standby...")
        if not self._check_health(self.standby):
            print("❌ Standby is not healthy!")
            self._set_read_write(self.primary)  # Rollback
            return False
        print("✓ Standby is healthy")

        # Step 5: Promote standby
        print("\n[Step 5/8] Promoting standby to primary...")
```

```
        self._promote_to_primary(self.standby)
        time.sleep(2)
        print("✓ Standby promoted")

        # Step 6: Update application configuration
        print("\n[Step 6/8] Updating application to use new primary...")
        self._update_app_config(self.standby['host'])
        print("✓ Application reconfigured")

        # Step 7: Verify traffic flowing
        print("\n[Step 7/8] Verifying traffic on new primary...")
        time.sleep(5)
        if self._verify_traffic(self.standby):
            print("✓ Traffic flowing to new primary")
        else:
            print("⚠️  Low traffic, please investigate")

        # Step 8: Demote old primary
        print("\n[Step 8/8] Demoting old primary to standby...")
        self._demote_to_standby(self.primary)
        print("✓ Old primary is now standby")

        # Swap roles
        self.primary, self.standby = self.standby, self.primary

        print("\n" + "="*60)
        print("    FAILOVER COMPLETED SUCCESSFULLY")
        print("="*60)
        print(f"New primary: {self.primary['host']}")
        print(f"New standby: {self.standby['host']}")

        return True

    def _verify_standby_caught_up(self):
        """Check replication lag"""
        # Query: SELECT pg_last_wal_receive_lsn() - pg_last_wal_replay_lsn()
        # If 0, standby is caught up
        print("Checking replication lag...")
        return True  # Simplified

    def _set_read_only(self, server):
        """Set database to read-only"""
        # ALTER SYSTEM SET default_transaction_read_only = on;
        print(f"Set {server['host']} to read-only")

    def _set_read_write(self, server):
        """Set database to read-write"""
```

```python
        # ALTER SYSTEM SET default_transaction_read_only = off;
        print(f"Set {server['host']} to read-write")

    def _check_health(self, server):
        """Verify server is healthy"""
        # SELECT 1;
        return True

    def _promote_to_primary(self, server):
        """Promote replica to primary"""
        # pg_ctl promote
        server['status'] = 'primary'

    def _demote_to_standby(self, server):
        """Demote primary to standby"""
        # Configure as replica
        server['status'] = 'standby'

    def _update_app_config(self, new_primary_host):
        """Update application to use new primary"""
        # Update configuration management (Consul, etcd)
        # Or update DNS
        print(f"Applications now connecting to {new_primary_host}")

    def _verify_traffic(self, server):
        """Check if traffic is flowing"""
        # Query connection count, queries per second
        return True

# Usage
controller = ManualFailoverController()

# Initiate planned failover (e.g., for maintenance)
controller.initiate_planned_failover()

# Output shows step-by-step progress:
# [Step 1/8] Pre-failover checks...
# ✓ Standby is caught up
# [Step 2/8] Setting primary to read-only mode...
# ...
```

6. Active-Active vs Active-Passive

Active-Passive (Traditional HA)

```
Architecture:

┌─────────────────────────────────────────────────┐
│                  │
│   ┌───────────────┐          │
│   │Virtual IP   │          │
│   │ 10.0.0.100  │          │
│   └───────┬───────┘          │
│      │          │
│      ↓          │
│   ┌───────────────┐  ┌───────────────┐ │
│   │  PRIMARY   │  │  STANDBY   │ │
│   │ (Active)   │──────→│ (Passive)  │ │
│   │      │  │      │ │
│   │ Handles ALL │  │ Does nothing │ │
│   │ traffic    │  │ (wasteful!) │ │
│   └───────────────┘  └───────────────┘ │
│                  │
└─────────────────────────────────────────────────┘

On Primary Failure:

┌─────────────────────────────────────────────────┐
│   ┌───────────────┐          │
│   │Virtual IP   │          │
│   │ 10.0.0.100  │ (moved)      │
│   └───────────────┬───────┘      │
│         │        │
│         ↓        │
│   ┌───────────────┐  ┌───────────────┐ │
│   │  PRIMARY   │  │  STANDBY   │ │
│   │ (Failed)   │  │(Now Active) │ │
│   │      │  │      │ │
│   │   ✗    │  │ Handles ALL │ │
│   │      │  │ traffic   │ │
│   └───────────────┘  └───────────────┘ │
└─────────────────────────────────────────────────┘

Failover Time:
1. Detect failure: 5-10 seconds
2. Activate standby: 5-10 seconds
3. Move virtual IP: 1-5 seconds
4. DNS propagation: 0-30 seconds
```

```
Total: 10-55 seconds downtime
```

Pros and Cons:

```
✓ ADVANTAGES:
  - Simpler to implement
  - No split-brain issues
  - Clear master/slave relationship
  - Easier to reason about
  - Lower cost (standby can be smaller instance)

✗ DISADVANTAGES:
  - Wasted resources (standby idle)
  - Downtime during failover (10-60 seconds)
  - Can't serve traffic during failover
  - Manual intervention may be needed

When to use:
- Stateful applications (databases)
- Budget constraints
- Simpler operations preferred
- Can tolerate brief downtime
```

Active-Active (Multi-Master)

```
Architecture:

┌─────────────────────────────────────────┐
│      Active-Active Setup                │
├─────────────────────────────────────────┤
│                                         │
│    ┌──────────────┐                     │
│    │Load Balancer │                     │
│    └──────┬───────┘                     │
│           │       │                     │
│    ┌──────┴───────┐                     │
│    ↓          ↓                         │
│  ┌─────────┐  ┌─────────┐               │
│  │SERVER 1 │  │SERVER 2 │               │
│  │(Active) │◄─►│(Active) │              │
│  │         │  │         │               │
│  │50% traffic│ │50% traffic│            │
│  │Read/Write│  │Read/Write│             │
│  └─────────┘  └─────────┘               │
```

```
│     Both serving traffic                │
│     Both can handle writes              │
└─────────────────────────────────────────┘

On Server 1 Failure:

┌─────────────────────────────────────────┐
│     ┌──────────────┐                    │
│     │Load Balancer │                    │
│     └──────┬───────┘                    │
│            │                            │
│            │ (detects failure)          │
│            ↓                            │
│  ┌──────────┐  ┌──────────┐             │
│  │SERVER 1  │  │SERVER 2  │             │
│  │ (Failed) │  │(Active)  │             │
│  │          │  │          │             │
│  │    ✗     │  │100% traffic│           │
│  │          │  │Read/Write│             │
│  └──────────┘  └──────────┘             │
│     Server 2 handles all traffic        │
│     No downtime! (instant failover)     │
└─────────────────────────────────────────┘

Failover Time: 0-5 seconds (just health check detection)
```

Implementation:

```
python
```

```python
class ActiveActiveCluster:
    def __init__(self, nodes):
        self.nodes = nodes
        self.health_status = {node['id']: True for node in nodes}

        # Start health monitoring
        threading.Thread(target=self._monitor_health, daemon=True).start()

    def _monitor_health(self):
        """Continuously check all nodes"""
        while True:
            for node in self.nodes:
                try:
                    response = requests.get(
                        f"{node['url']}/health",
                        timeout=2
                    )

                    previous_status = self.health_status[node['id']]
                    current_status = (response.status_code == 200)

                    self.health_status[node['id']] = current_status

                    # Log status changes
                    if previous_status and not current_status:
                        print(f"🚨 Node {node['id']} went DOWN")
                        self._alert_ops(f"Node {node['id']} is unhealthy")
                    elif not previous_status and current_status:
                        print(f"✅ Node {node['id']} is UP again")
                        self._alert_ops(f"Node {node['id']} recovered")

                except Exception as e:
                    self.health_status[node['id']] = False

            time.sleep(5)

    def get_healthy_nodes(self):
        """Get list of currently healthy nodes"""
        return [
            node for node in self.nodes
            if self.health_status[node['id']]
        ]

    def route_request(self, request):
        """Route to any healthy node (load balanced)"""
        healthy = self.get_healthy_nodes()
```

```
        if not healthy:
            raise Exception("No healthy nodes available!")

        # Round-robin or random selection
        node = healthy[hash(request) % len(healthy)]

        try:
            response = requests.post(
                f"{node['url']}/api/process",
                json=request,
                timeout=5
            )
            return response.json()

        except Exception as e:
            # This node failed, mark unhealthy
            self.health_status[node['id']] = False

            # Retry with another node
            healthy.remove(node)
            if healthy:
                print(f"Retrying with different node...")
                return self.route_request(request)
            else:
                raise Exception("All nodes failed")

    def _alert_ops(self, message):
        print(f"📧 {message}")

# Usage
cluster = ActiveActiveCluster([
    {'id': 'node1', 'url': 'http://server1:8000'},
    {'id': 'node2', 'url': 'http://server2:8000'},
    {'id': 'node3', 'url': 'http://server3:8000'}
])

# All nodes serve traffic
# If node1 fails, traffic automatically goes to node2 and node3
# Zero downtime!

for i in range(100):
    result = cluster.route_request({'task': f'task-{i}'})
```

Active-Active Database (Complex!)

```
Challenge: Both databases accepting writes

┌────────────────┐     ┌────────────────┐
│  DB 1          │◄───►│  DB 2          │
│ (Active)       │ Sync│ (Active)       │
└────────────────┘     └────────────────┘
   ↑                  ↑
   │                  │
 App writes         App writes

Problem: Write conflicts!

Time   DB 1                DB 2
─────────────────────────────────────────
10:00  User balance: $1000  User balance: $1000
10:01  Withdraw $500        Withdraw $600
       Balance = $500       Balance = $400
10:02  Sync to DB2 →        ← Sync to DB1
       Conflict! Which is correct?
```

Conflict Resolution Strategies:

```python
```

```python
# Strategy 1: Last Write Wins (LWW)
class LastWriteWins:
    """Use timestamp to resolve conflicts"""

    def resolve_conflict(self, value1, value2):
        # Each value has timestamp
        if value1['timestamp'] > value2['timestamp']:
            return value1
        else:
            return value2

# Example:
# DB1: {balance: 500, timestamp: 10:01:30}
# DB2: {balance: 400, timestamp: 10:01:45}
# Winner: DB2 (later timestamp)
# Problem: Lost $500 withdrawal!

# Strategy 2: Application-Level Conflict Resolution
class BankingConflictResolver:
    """Custom logic for banking domain"""

    def resolve_conflict(self, db1_state, db2_state):
        # Get transaction logs from both
        db1_transactions = db1_state['transactions']
        db2_transactions = db2_state['transactions']

        # Merge transactions
        all_transactions = self._merge_transactions(
            db1_transactions,
            db2_transactions
        )

        # Recalculate balance from scratch
        final_balance = db1_state['initial_balance']
        for tx in all_transactions:
            if tx['type'] == 'deposit':
                final_balance += tx['amount']
            elif tx['type'] == 'withdraw':
                final_balance -= tx['amount']

        return final_balance

    def _merge_transactions(self, tx1, tx2):
        """Merge and deduplicate transactions"""
        merged = {}
```

```python
        for tx in tx1 + tx2:
            merged[tx['id']] = tx

        # Sort by timestamp
        return sorted(merged.values(), key=lambda x: x['timestamp'])

# Strategy 3: CRDTs (Conflict-Free Replicated Data Types)
class CounterCRDT:
    """Increment-only counter that never conflicts"""

    def __init__(self, node_id):
        self.node_id = node_id
        self.counts = {}  # node_id -> count

    def increment(self, amount=1):
        """Increment counter on this node"""
        if self.node_id not in self.counts:
            self.counts[self.node_id] = 0

        self.counts[self.node_id] += amount

    def merge(self, other_counts):
        """Merge counts from other node"""
        for node_id, count in other_counts.items():
            if node_id not in self.counts:
                self.counts[node_id] = 0

            # Take maximum (both nodes can increment)
            self.counts[node_id] = max(self.counts[node_id], count)

    def get_total(self):
        """Get total count across all nodes"""
        return sum(self.counts.values())

# Usage:
# Node 1: counter.increment(5)  → counts = {node1: 5}
# Node 2: counter.increment(3)  → counts = {node2: 3}
# After sync: counts = {node1: 5, node2: 3}, total = 8
# No conflicts!
```

Active-Active Comparison

```
┌────────────────────────┬───────────────────┬────────────────────────┐
│ Feature    │ Active-Passive │ Active-Active │
├────────────────────────┼───────────────────┼────────────────────────┤
```

Resource Usage	50% (standby idle)	100% (all serve)
Failover Time	10-60 seconds	0-5 seconds
Complexity	Simple	Complex
Consistency	Strong	Eventual
Write Conflicts	None	Possible
Cost Efficiency	50%	100%
Geographic Dist	Limited	Excellent
Split-Brain Risk	Low	Higher

Decision:
- Active-Passive: Databases, stateful apps
- Active-Active: Web servers, stateless apps, multi-region

Multi-Region Active-Active

Global Active-Active Architecture:

```
┌─────────────────────────────────────────┐
│        Global Active-Active Setup        │
├─────────────────────────────────────────┤
│                                          │
│  US Region (Active)     EU Region (Active) │
│  ┌─────────────────┐   ┌──────────────┐  │
│  │ Load Balancer   │   │ Load Balancer │ │
│  └────────┬────────┘   └──────┬───────┘  │
│
```