Chapter 4: Load Balancing

1. What is Load Balancing and Why We Need It

Definition

Load Balancing is the process of distributing network traffic across multiple servers to ensure no single server becomes overwhelmed.

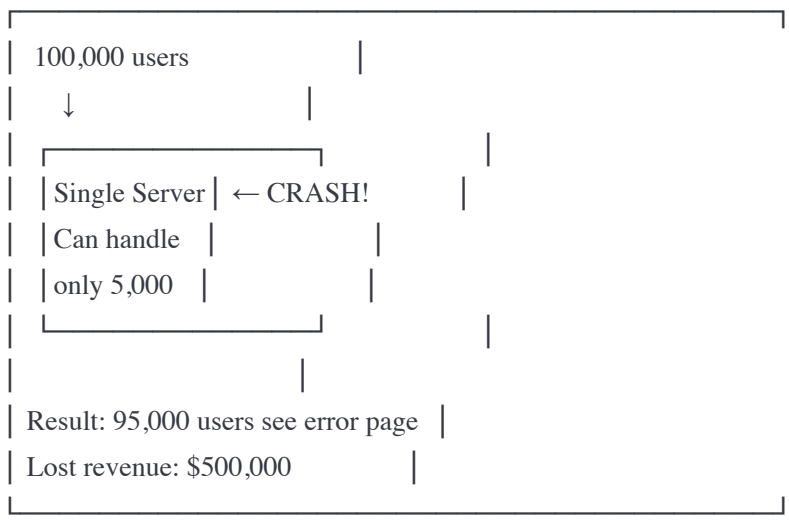
Analogy: Think of a grocery store with multiple checkout lanes.

Why We Need Load Balancing

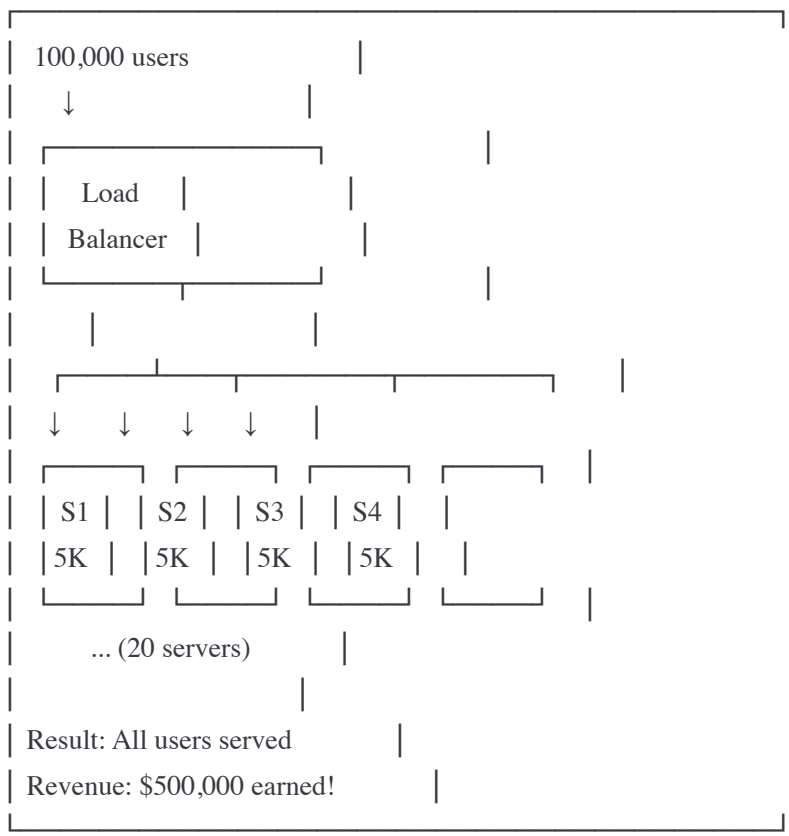
Problem 1: Single Server Overload

Scenario: E-commerce website during flash sale

Without Load Balancer:

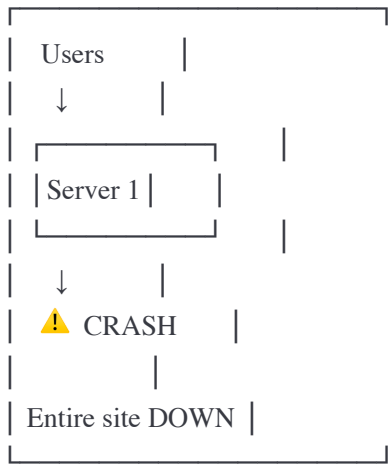


With Load Balancer:

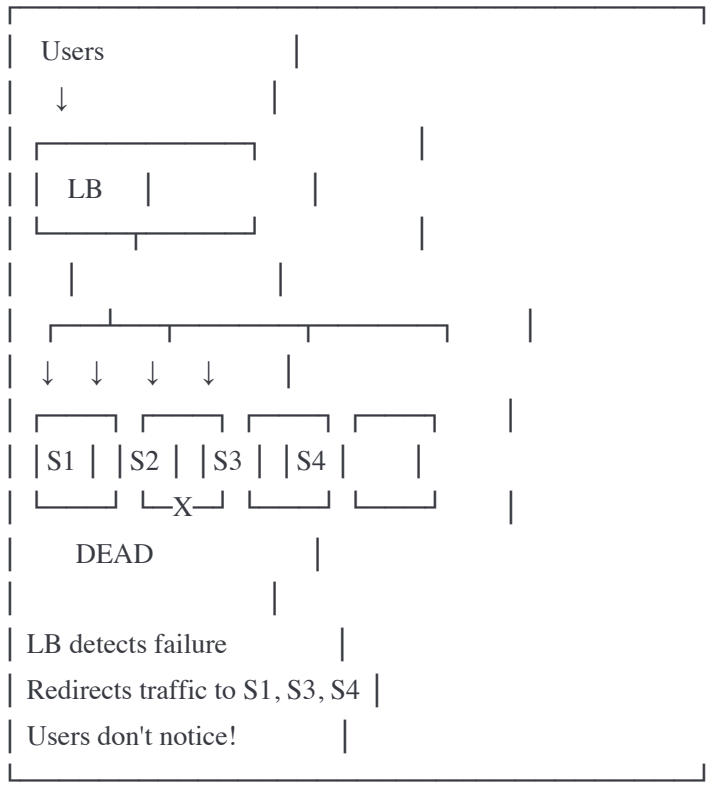


Problem 2: Single Point of Failure

Without Load Balancer:



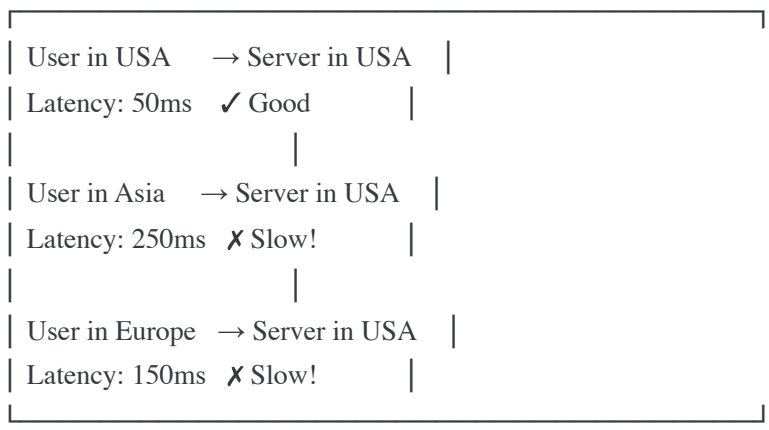
With Load Balancer + Multiple Servers:



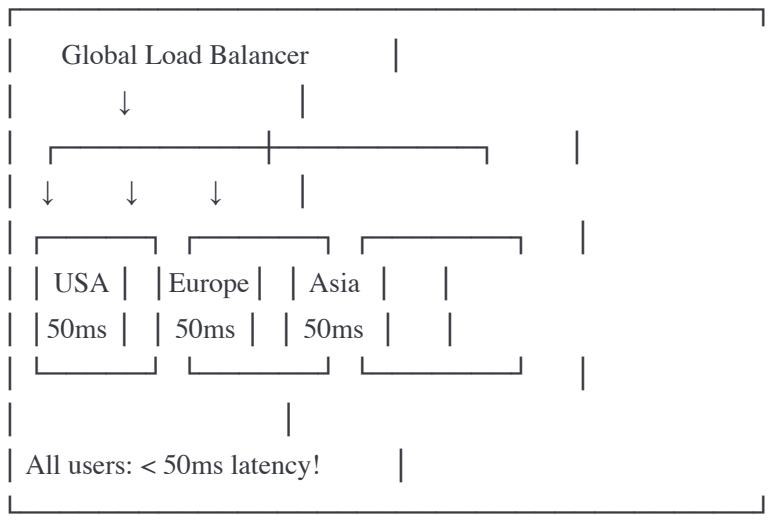
Problem 3: Geographic Distribution

Users from different locations:

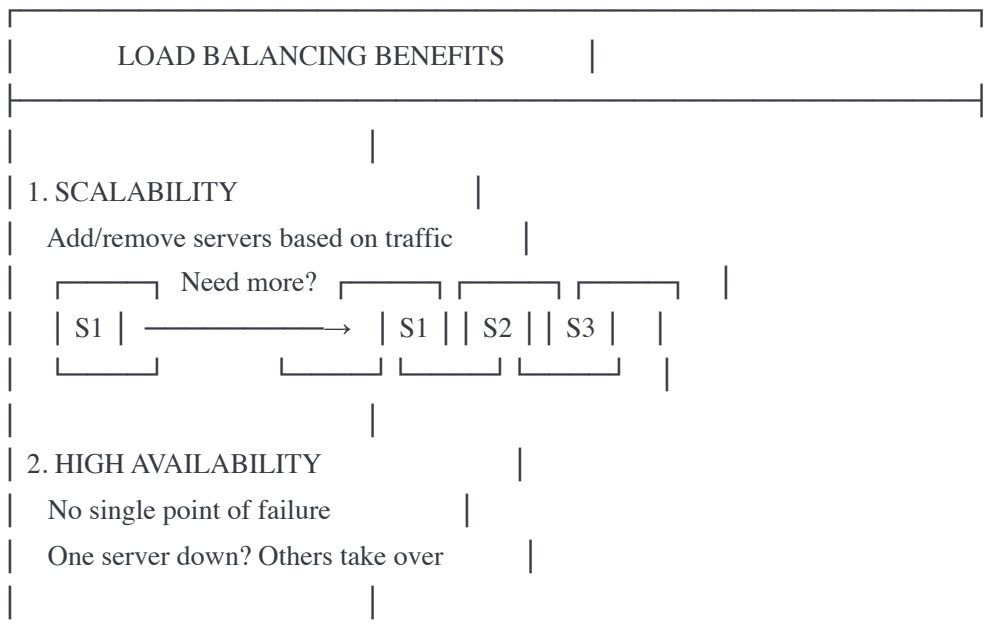
Without Load Balancing:

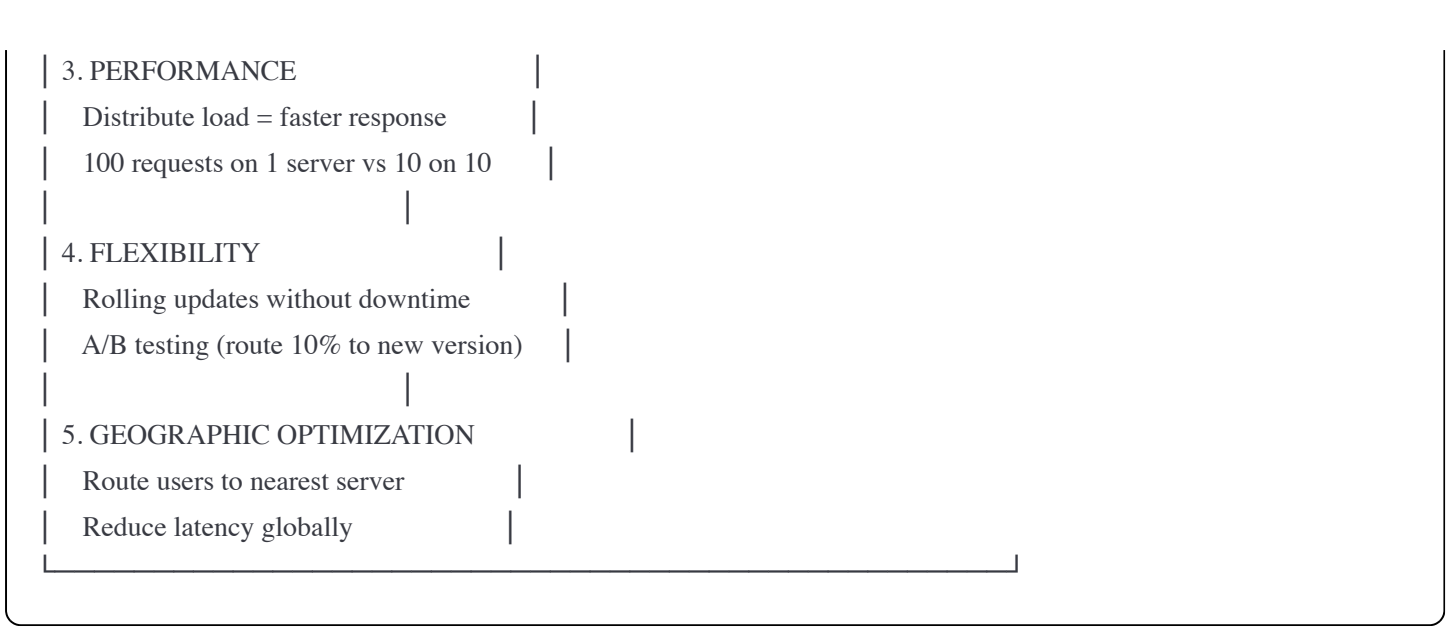


With Geographic Load Balancing:



Benefits of Load Balancing





---

2. Load Balancing Algorithms

Load balancers decide **which server gets the next request**. Different algorithms suit different scenarios.

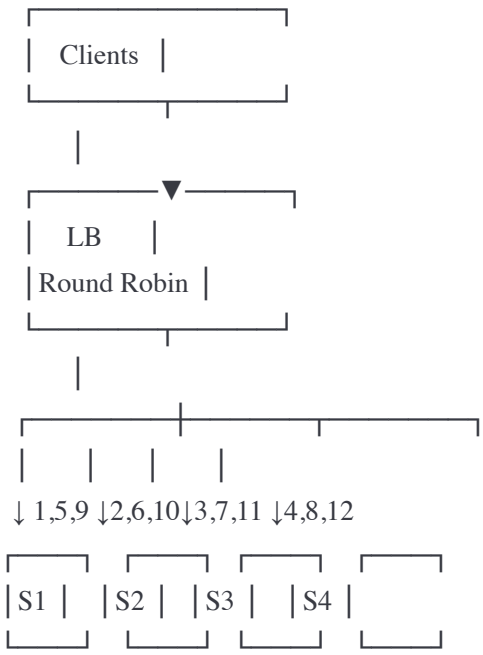
1. Round Robin - The Simple Approach

How it works: Distribute requests evenly in sequence

Round Robin Distribution:

Request 1 → Server 1
Request 2 → Server 2
Request 3 → Server 3
Request 4 → Server 4
Request 5 → Server 1 (back to first)
Request 6 → Server 2
...and so on

Visual:



Python Implementation:

python

```python
class RoundRobinLoadBalancer:
    def __init__(self, servers):
        self.servers = servers
        self.current_index = 0

    def get_next_server(self):
        """Get next server in round-robin fashion"""
        server = self.servers[self.current_index]

        # Move to next server for next request
        self.current_index = (self.current_index + 1) % len(self.servers)

        return server

    def handle_request(self, request):
        server = self.get_next_server()
        print(f"Routing request to {server}")
        return server

# Example usage
lb = RoundRobinLoadBalancer([
    'server1.com',
    'server2.com',
    'server3.com',
    'server4.com'
])

# Simulate 10 requests
for i in range(10):
    lb.handle_request(f"request_{i}")

# Output:
# Routing request to server1.com
# Routing request to server2.com
# Routing request to server3.com
# Routing request to server4.com
# Routing request to server1.com
# Routing request to server2.com
# ...
```

Pros and Cons:

✓ ADVANTAGES:
- Simple to implement
- Fair distribution
- No server state needed
- Works well when all servers are equal

✗ DISADVANTAGES:
- Ignores server load
- Assumes all requests take same time
- No consideration for server capacity
- Can overload slow servers

Example Problem:
Server 1: Fast (handles request in 10ms)
Server 2: Fast (handles request in 10ms)
Server 3: Slow (handles request in 100ms)

Round robin treats them equally!
Server 3 gets overloaded while 1 & 2 are idle.

When to use:

- All servers have similar capacity
- Requests are roughly uniform in complexity
- Simplicity is priority

---

2. Weighted Round Robin - Respecting Capacity

How it works: Distribute requests based on server capacity

Server Weights:
Server 1: weight = 4 (powerful server)
Server 2: weight = 2 (medium server)
Server 3: weight = 1 (weak server)

Distribution pattern:
Request 1 → Server 1
Request 2 → Server 1
Request 3 → Server 1
Request 4 → Server 1
Request 5 → Server 2
Request 6 → Server 2
Request 7 → Server 3
Request 8 → Server 1 (cycle repeats)

Out of 7 requests:
- Server 1 gets 4 (57%)
- Server 2 gets 2 (29%)
- Server 3 gets 1 (14%)

Python Implementation:

python

```python
class WeightedRoundRobinLoadBalancer:
    def __init__(self, servers_with_weights):
        """
        servers_with_weights: [('server1', 4), ('server2', 2), ('server3', 1)]
        """
        self.servers = []

        # Expand servers based on weights
        for server, weight in servers_with_weights:
            self.servers.extend([server] * weight)

        self.current_index = 0

    def get_next_server(self):
        server = self.servers[self.current_index]
        self.current_index = (self.current_index + 1) % len(self.servers)
        return server

# Example usage
lb = WeightedRoundRobinLoadBalancer([
    ('high-performance-server', 5),
    ('medium-server', 3),
    ('low-end-server', 1)
])

# Track distribution
distribution = {}
for i in range(100):
    server = lb.get_next_server()
    distribution[server] = distribution.get(server, 0) + 1

print(distribution)
# Output:
# {
#   'high-performance-server': 56,  (5/9 = 55.6%)
#   'medium-server': 33,            (3/9 = 33.3%)
#   'low-end-server': 11            (1/9 = 11.1%)
# }
```

When to use:

- Servers have different capacities
- You know server capabilities beforehand
- Static server configuration

---

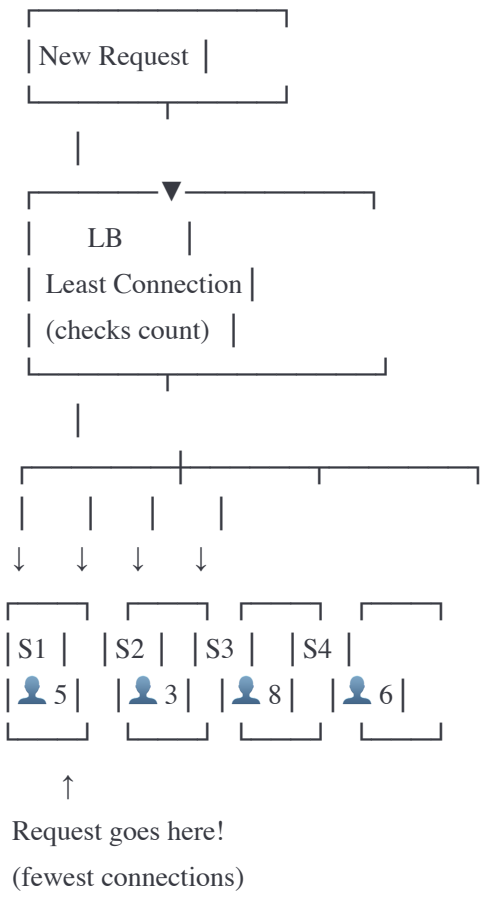
3. Least Connections - The Smart Choice

How it works: Send request to server with fewest active connections

Current State:
Server 1: 5 active connections
Server 2: 3 active connections ← Next request goes here!
Server 3: 8 active connections
Server 4: 6 active connections

Why this matters:
Connection = ongoing request/session
More connections = more load
Route to least loaded server

Visual:



Python Implementation:

python

```python
import threading
from collections import defaultdict

class LeastConnectionsLoadBalancer:
    def __init__(self, servers):
        self.servers = servers
        self.connections = defaultdict(int)
        self.lock = threading.Lock()

    def get_next_server(self):
        """Get server with least active connections"""
        with self.lock:
            # Find server with minimum connections
            server = min(self.servers, key=lambda s: self.connections[s])

            # Increment connection count
            self.connections[server] += 1

            return server

    def release_connection(self, server):
        """Called when request completes"""
        with self.lock:
            self.connections[server] = max(0, self.connections[server] - 1)

    def get_stats(self):
        """View current load distribution"""
        return dict(self.connections)

# Example usage
lb = LeastConnectionsLoadBalancer([
    'server1.com',
    'server2.com',
    'server3.com'
])

# Simulate requests
def handle_request(request_id, duration):
    server = lb.get_next_server()
    print(f"Request {request_id} → {server}")

    # Simulate work
    time.sleep(duration)

    # Release when done
    lb.release_connection(server)
```

```
    print(f"Request {request_id} completed on {server}")

# Request 1 takes 5 seconds
threading.Thread(target=handle_request, args=(1, 5)).start()

# Small delay
time.sleep(0.1)

# Requests 2 and 3 are quick
threading.Thread(target=handle_request, args=(2, 1)).start()
threading.Thread(target=handle_request, args=(3, 1)).start()

# Check distribution
time.sleep(0.5)
print(f"Current connections: {lb.get_stats()}")

# Output shows server1 is busy (5s request)
# so requests 2 & 3 go to server2 and server3
```

Real-World Example:

Web Application with Varying Request Times:

Scenario:
- Request A: Simple page load (50ms)
- Request B: Simple page load (50ms)
- Request C: Complex report generation (10 seconds)
- Request D: Simple page load (50ms)

Round Robin (bad):
Time 0: A → Server 1 (50ms)
Time 50: B → Server 2 (50ms)
Time 100: C → Server 3 (10s, blocks server!)
Time 150: D → Server 1 (50ms)
Result: Server 3 tied up for 10s with one request!

Least Connections (good):
Time 0: A → Server 1 (50ms)
Time 50: B → Server 2 (50ms)
Time 100: C → Server 3 (10s)
Time 150: D → Server 1 (Server 3 still busy!)
Result: Server 1 and 2 handle new requests while 3 works on heavy task

Pros and Cons:

✓ ADVANTAGES:
- Adapts to varying request times
- Better distribution under real load
- Prevents overloading slow requests
- Handles long-lived connections well

✗ DISADVANTAGES:
- More complex to implement
- Requires tracking state
- Slight overhead for counting
- May not account for actual server load (CPU, memory)

When to use:
- Long-lived connections (WebSockets, streaming)
- Requests vary significantly in processing time
- Database connection pooling

---

4. Weighted Least Connections

Combines both: Consider both connection count AND server capacity

```
python
```

```python
class WeightedLeastConnectionsLoadBalancer:
    def __init__(self, servers_with_weights):
        """
        servers_with_weights: [('server1', 5), ('server2', 3), ('server3', 1)]
        Higher weight = more capacity
        """
        self.servers = dict(servers_with_weights)
        self.connections = defaultdict(int)
        self.lock = threading.Lock()

    def get_next_server(self):
        """Get server with best ratio of connections to capacity"""
        with self.lock:
            # Calculate load ratio: connections / weight
            # Lower ratio = less loaded relative to capacity
            server = min(
                self.servers.keys(),
                key=lambda s: self.connections[s] / self.servers[s]
            )

            self.connections[server] += 1
            return server

    def release_connection(self, server):
        with self.lock:
            self.connections[server] = max(0, self.connections[server] - 1)

# Example
lb = WeightedLeastConnectionsLoadBalancer([
    ('powerful-server', 10),    # Can handle 10x baseline
    ('medium-server', 5),       # Can handle 5x baseline
    ('weak-server', 2)          # Can handle 2x baseline
])

# After some requests:
# powerful-server: 40 connections → ratio: 40/10 = 4.0
# medium-server: 25 connections   → ratio: 25/5 = 5.0
# weak-server: 6 connections      → ratio: 6/2 = 3.0

# Next request goes to weak-server (lowest ratio!)
```

---

5. IP Hash - Session Persistence

How it works: Same client always goes to same server (based on IP address)

Why needed: Sessions/Shopping Carts

Problem without IP Hash:
User adds item to cart → Server 1 (cart saved in Server 1's memory)
User views cart → Server 2 (Server 2 doesn't know about cart!)
Result: Cart appears empty! ❌

Solution with IP Hash:
User IP: 192.168.1.100
Hash: hash(192.168.1.100) % 4 = 2
All requests from this IP → Server 2
Result: Cart persists! ✓

Visual:
User 1 (IP: 192.168.1.5)
↓ hash → Server 1
All requests → Server 1

User 2 (IP: 192.168.1.10)
↓ hash → Server 3
All requests → Server 3

User 3 (IP: 192.168.1.15)
↓ hash → Server 2
All requests → Server 2

Python Implementation:

python

```python
import hashlib

class IPHashLoadBalancer:
    def __init__(self, servers):
        self.servers = servers

    def get_server_for_ip(self, client_ip):
        """Consistently map IP to same server"""
        # Hash the IP address
        hash_value = int(hashlib.md5(client_ip.encode()).hexdigest(), 16)

        # Modulo to get server index
        server_index = hash_value % len(self.servers)

        return self.servers[server_index]

    def handle_request(self, client_ip, request):
        server = self.get_server_for_ip(client_ip)
        print(f"Client {client_ip} → {server}")
        return server

# Example usage
lb = IPHashLoadBalancer([
    'server1.com',
    'server2.com',
    'server3.com',
    'server4.com'
])

# Same IP always goes to same server
for _ in range(5):
    lb.handle_request('192.168.1.100', 'request')

# Output (all go to same server):
# Client 192.168.1.100 → server3.com
# Client 192.168.1.100 → server3.com
# Client 192.168.1.100 → server3.com
# Client 192.168.1.100 → server3.com
# Client 192.168.1.100 → server3.com

# Different IPs go to different servers
lb.handle_request('192.168.1.50', 'request')   # → server1.com
lb.handle_request('192.168.1.75', 'request')   # → server2.com
lb.handle_request('192.168.1.200', 'request')  # → server4.com
```

Pros and Cons:

✓ ADVANTAGES:
- Session persistence without shared storage
- No session data synchronization needed
- Simple to implement
- Consistent user experience

✗ DISADVANTAGES:
- Uneven distribution if IPs not random
- If server fails, sessions lost
- Users behind NAT get same server
- Adding/removing servers changes mappings

When to use:
- Stateful applications (sessions in memory)
- No shared session storage
- User experience requires consistency

---

6. Least Response Time

How it works: Route to server with fastest response time

python

```python
import time
from collections import defaultdict

class LeastResponseTimeLoadBalancer:
    def __init__(self, servers):
        self.servers = servers
        self.response_times = defaultdict(lambda: [])
        self.lock = threading.Lock()

    def get_next_server(self):
        """Get server with lowest average response time"""
        with self.lock:
            # Calculate average response time for each server
            server_averages = {}

            for server in self.servers:
                times = self.response_times[server]
                if times:
                    # Average of last 10 requests
                    recent = times[-10:]
                    server_averages[server] = sum(recent) / len(recent)
                else:
                    # No history, assume fast
                    server_averages[server] = 0

            # Return fastest server
            return min(server_averages, key=server_averages.get)

    def record_response(self, server, response_time_ms):
        """Record response time after request completes"""
        with self.lock:
            self.response_times[server].append(response_time_ms)

            # Keep only last 100 measurements
            if len(self.response_times[server]) > 100:
                self.response_times[server].pop(0)

# Example usage
lb = LeastResponseTimeLoadBalancer([
    'fast-server.com',
    'medium-server.com',
    'slow-server.com'
])

# Simulate requests with different response times
lb.record_response('fast-server.com', 10)
```

```
lb.record_response('medium-server.com', 50)
lb.record_response('slow-server.com', 200)

# Next request goes to fast-server.com
next_server = lb.get_next_server()
print(f"Routing to: {next_server}")  # → fast-server.com
```

Algorithm Comparison Table

Algorithm	Complexity	Session Safe?	Load Aware?	Best For
Round Robin	O(1)	No	No	Simple apps
Weighted RR	O(1)	No	Partial (static)	Mixed capacity
Least Connections	O(n)	No	Yes (dynamic)	Long-lived connections
IP Hash	O(1)	Yes	No	Stateful apps
Least Response Time	O(n)	No	Yes (adaptive)	Performance-critical
Random	O(1)	No	No	Simple stateless

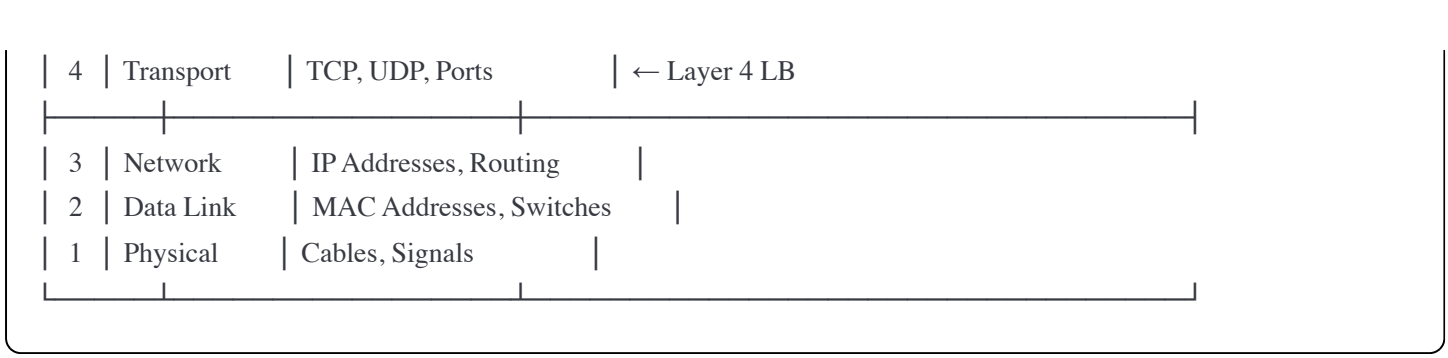
---

3. Layer 4 vs Layer 7 Load Balancing

Understanding the OSI model layers helps explain these concepts.

OSI Model Quick Reference

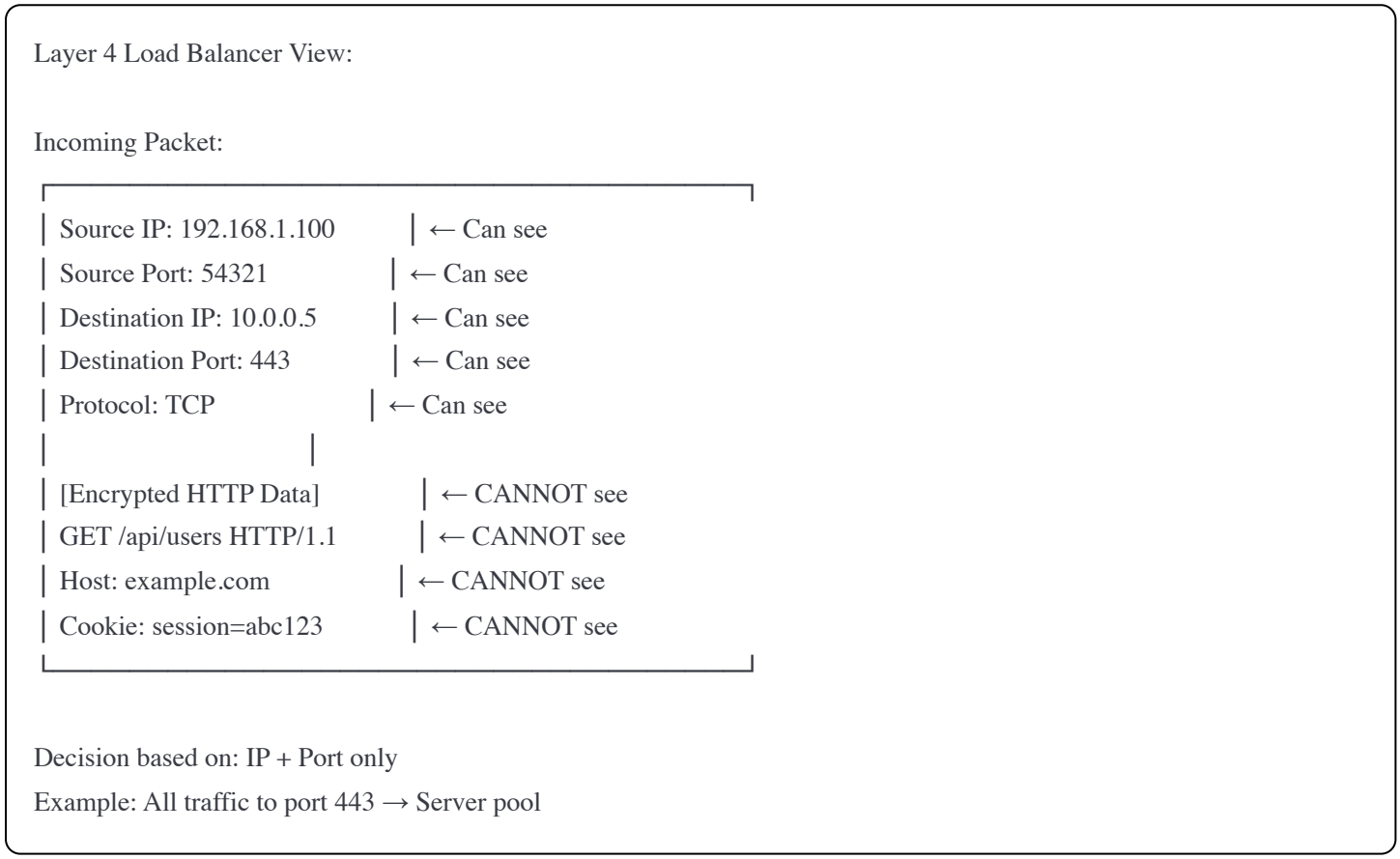
OSI MODEL (7 Layers)		
7	Application	HTTP, HTTPS, FTP, SMTP
6	Presentation	SSL/TLS, Encryption
5	Session	Sessions, Connections



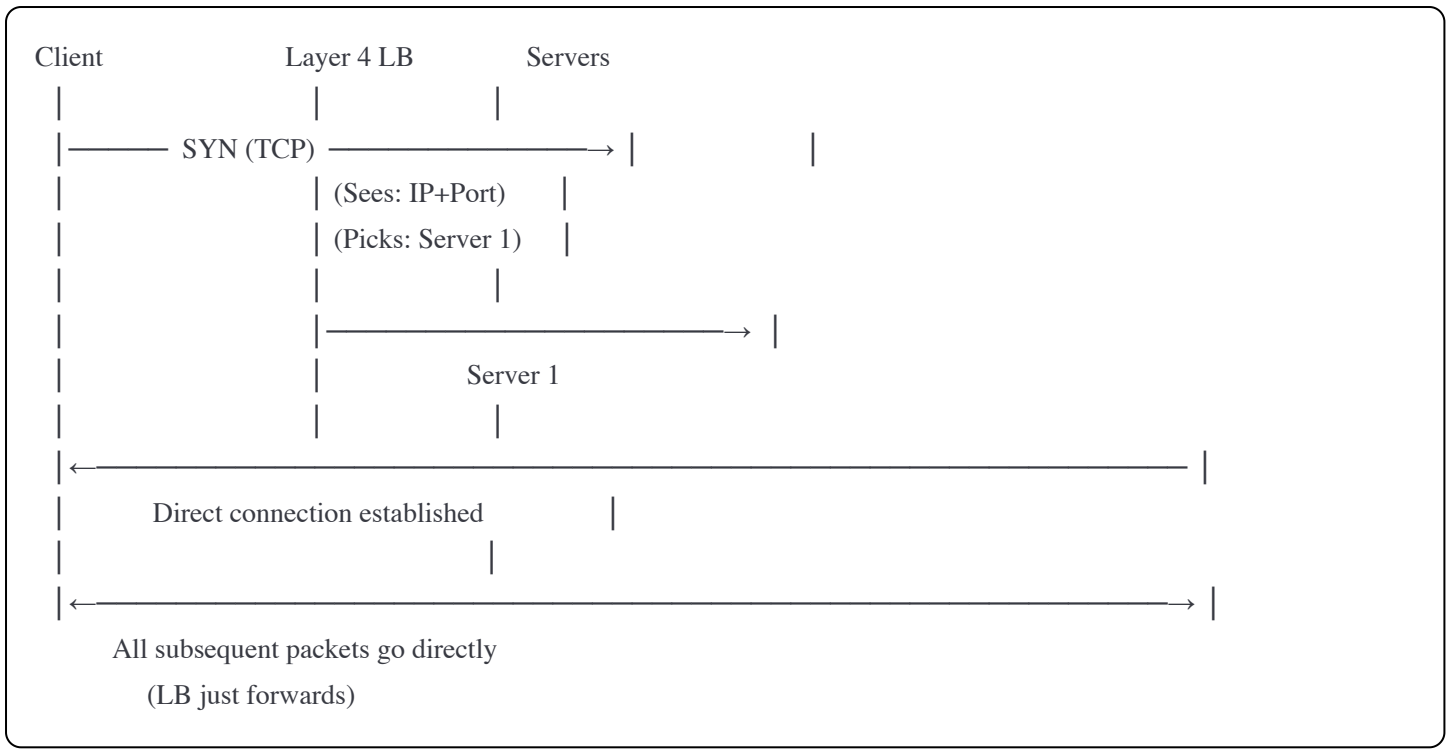
Layer 4 Load Balancing (Transport Layer)

What it sees: IP addresses, ports, TCP/UDP

What it DOESN'T see: HTTP headers, URLs, cookies



How it works:



Configuration Example (HAProxy):

```
# Layer 4 Load Balancer Configuration
frontend tcp_front
    bind *:80
    mode tcp                # Layer 4 mode
    default_backend tcp_back

backend tcp_back
    mode tcp
    balance roundrobin      # Simple algorithm

    server server1 10.0.0.10:80 check
    server server2 10.0.0.11:80 check
    server server3 10.0.0.12:80 check
```

Characteristics:

✓ ADVANTAGES:
- FAST (minimal processing)
- Low latency (just forwards packets)
- Can handle any protocol (TCP/UDP)
- Simple configuration
- High throughput
- Works with encrypted traffic (TLS/SSL)

✗ DISADVANTAGES:
- Cannot route based on content
- No URL-based routing
- Cannot inspect HTTP headers
- Limited to IP/Port decisions
- No application-aware features

Performance:
Throughput: 1-10 Gbps per instance
Latency overhead: < 1ms

Layer 7 Load Balancing (Application Layer)

What it sees: Everything! HTTP headers, URLs, cookies, request body

Layer 7 Load Balancer View:

Incoming Request:

```
┌──────────────────────────────────────────────────┐
│ GET /api/users/123 HTTP/1.1      │ ← Can see & route on
│ Host: api.example.com            │ ← Can see & route on
│ Cookie: session=abc123           │ ← Can see & route on
│ User-Agent: Mobile App           │ ← Can see & route on
│ Content-Type: application/json   │ ← Can see & route on
│                                  │
│ {                                │ ← Can see & parse
│   "name": "John",                │
│   "email": "john@example.com"    │
│ }                                │
└──────────────────────────────────────────────────┘
```

Can make intelligent routing decisions:
- /api/* → API servers
- /static/* → CDN
- Mobile app → Mobile-optimized servers
- Premium users → High-performance servers

Advanced Routing Examples:

1. URL-Based Routing:
   /api/* → API server pool
   /images/* → Image server pool
   /videos/* → Video server pool

2. Host-Based Routing:
   api.example.com → API servers
   www.example.com → Web servers
   admin.example.com → Admin servers

3. Content-Based Routing:
   JSON requests → JSON API servers
   XML requests → Legacy SOAP servers

4. User-Based Routing:
   Premium users (from cookie) → Fast servers
   Free users → Standard servers

5. Geographic Routing:
   User-Country: US → US servers
   User-Country: EU → EU servers

6. A/B Testing:
   10% of users → New version
   90% of users → Stable version

Configuration Example (NGINX):

nginx

```
# Layer 7 Load Balancer Configuration

# Define server pools
upstream api_servers {
    least_conn;                    # Smart algorithm
    server api1.example.com:8000;
    server api2.example.com:8000;
    server api3.example.com:8000;
}

upstream web_servers {
    server web1.example.com:80;
    server web2.example.com:80;
}

upstream static_servers {
    server cdn1.example.com:80;
    server cdn2.example.com:80;
}

server {
    listen 80;
    server_name example.com;

    # Route based on URL path
    location /api/ {
        proxy_pass http://api_servers;

        # Add custom headers
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
    }

    location /static/ {
        proxy_pass http://static_servers;

        # Cache static content
        proxy_cache my_cache;
        proxy_cache_valid 200 1h;
    }

    location / {
        proxy_pass http://web_servers;
    }
}
```

```
# Advanced: Route based on cookie
map $cookie_user_tier $backend {
    premium     premium_servers;
    default     standard_servers;
}
```

Real-World Example: Microservices Architecture

Request: GET /orders/123

Layer 7 LB analyzes:

```
┌─────────────────────────────────────────────┐
│ URL: /orders/123                            │
│                                             │
│ Routing Logic:                              │
│                                             │
│ IF path starts with /orders                 │
│   → Order Service (port 8001)               │
│                                             │
│ IF path starts with /users                  │
│   → User Service (port 8002)                │
│                                             │
│ IF path starts with /payments               │
│   → Payment Service (port 8003)             │
│                                             │
│ IF path starts with /catalog                │
│   → Catalog Service (port 8004)             │
└─────────────────────────────────────────────┘
```

Result: Routes to Order Service cluster

Layer 7 Features:

python

```python
# Example: Custom routing logic in Layer 7 LB

class Layer7LoadBalancer:
    def route_request(self, request):
        # URL-based routing
        if request.path.startswith('/api/v1/'):
            return self.route_to('api_v1_servers')

        if request.path.startswith('/api/v2/'):
            return self.route_to('api_v2_servers')

        # Header-based routing
        if request.headers.get('X-Mobile-App'):
            return self.route_to('mobile_optimized_servers')

        # Cookie-based routing (A/B testing)
        if request.cookies.get('beta_tester') == 'true':
            return self.route_to('beta_servers')

        # Geographic routing
        country = request.headers.get('CloudFront-Viewer-Country')
        if country == 'US':
            return self.route_to('us_servers')
        elif country in ['UK', 'FR', 'DE']:
            return self.route_to('eu_servers')

        # Default
        return self.route_to('default_servers')
```

Layer 4 vs Layer 7 Comparison

Feature	Layer 4	Layer 7
Speed	Very Fast	Slower
Latency	< 1ms	2-10ms
Throughput	10+ Gbps	1-5 Gbps
Can see	IP, Port	Everything
Routing basis	IP/Port	URL, Headers
Protocol support	Any TCP/UDP	HTTP/HTTPS
SSL termination	No	Yes
Content caching	No	Yes

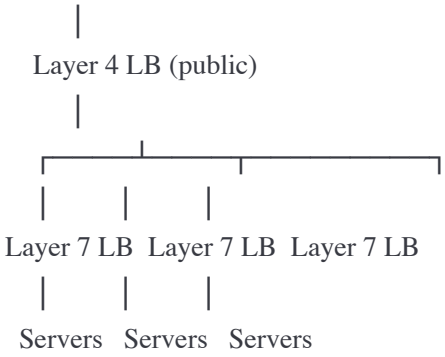
Compression	No	Yes
URL rewriting	No	Yes
Best for	High throughput	Smart routing
	Any protocol	Microservices
	Simple routing	A/B testing
Cost	Lower	Higher
Complexity	Simple	Complex

When to use each:

Use Layer 4 when:
✓ Maximum performance needed
✓ Non-HTTP protocols (databases, game servers)
✓ Simple round-robin sufficient
✓ Budget constraints
✓ Traffic is encrypted end-to-end

Use Layer 7 when:
✓ Need intelligent routing
✓ Microservices architecture
✓ Want to cache content
✓ Need SSL termination
✓ A/B testing required
✓ URL-based routing needed

Common Pattern: Use BOTH



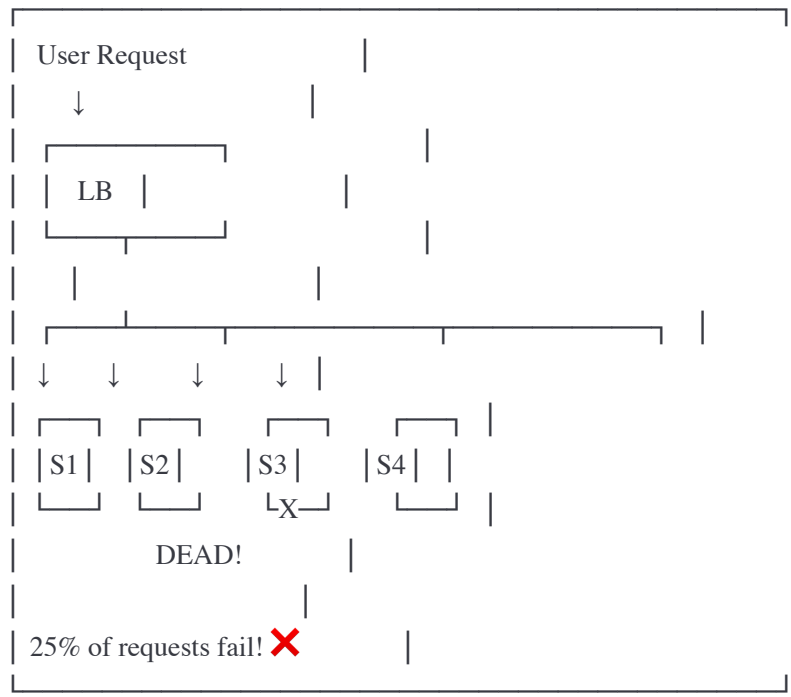
---

4. Health Checks and Failure Detection

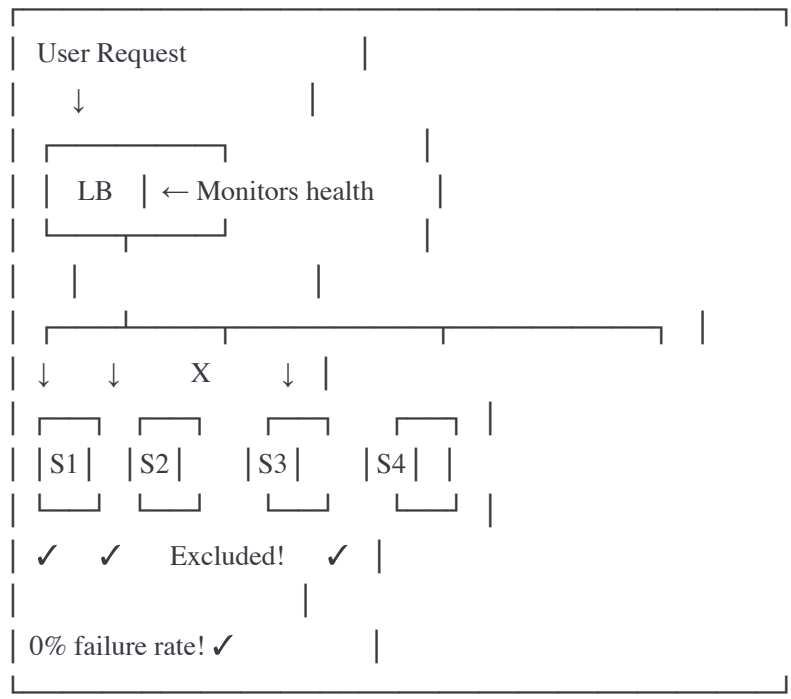
Critical Feature: Load balancer must know which servers are healthy!

Why Health Checks Matter

Without Health Checks:



With Health Checks:



Types of Health Checks

1. Passive Health Checks (Monitor Real Traffic)

How it works:

- Monitor actual user requests
- If request fails → mark server as unhealthy
- After N failures → remove from pool

Passive Monitoring:

User → LB → Server 1 (Success) ✓
User → LB → Server 2 (Success) ✓
User → LB → Server 3 (Timeout) ✗
User → LB → Server 3 (Error 500) ✗
User → LB → Server 3 (Error 500) ✗
↓
After 3 failures: Remove Server 3

Advantages:

- No extra traffic
- Detects real problems users experience

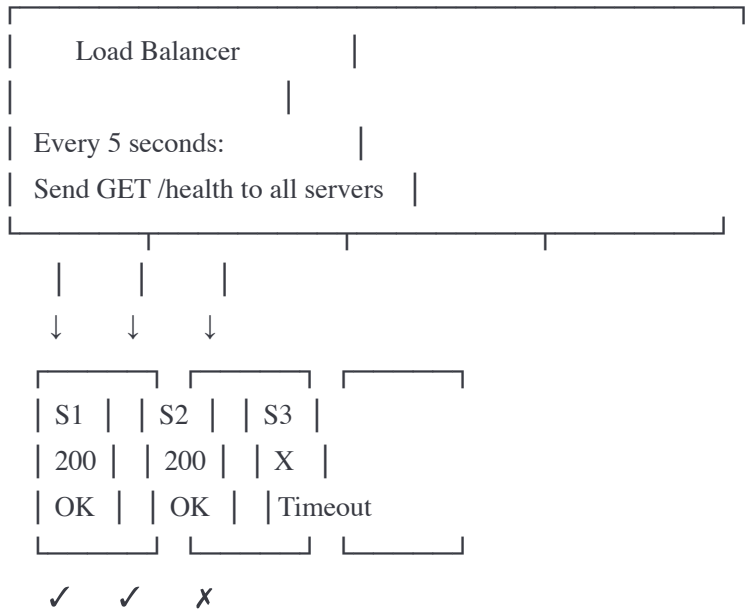
Disadvantages:

- Users might experience failures
- Slower to detect (needs multiple failures)

2. Active Health Checks (Proactive Monitoring)

How it works:
- LB sends test requests to servers
- Checks response periodically
- Removes unresponsive servers immediately

Active Health Check:



Result: S3 removed before any user sees failure!

Advantages:
- Detects failures before users affected
- Faster detection
- Consistent monitoring

Disadvantages:
- Extra network traffic
- Load on servers from health checks

Health Check Configuration Examples

Basic HTTP Health Check

nginx

```
# NGINX Configuration
upstream backend {
    server backend1.example.com;
    server backend2.example.com;
    server backend3.example.com;

    # Health check settings
    check interval=3000      # Check every 3 seconds
          rise=2             # 2 successful checks = healthy
          fall=3             # 3 failed checks = unhealthy
          timeout=1000       # Timeout after 1 second
          type=http;         # HTTP health check

    check_http_send "GET /health HTTP/1.0\r\n\r\n";
    check_http_expect_alive http_2xx http_3xx;
}
```

Advanced Health Check

```
# HAProxy Configuration
backend app_servers
    balance roundrobin

    # Basic health check
    option httpchk GET /health

    # Server definitions with health check parameters
    server app1 10.0.0.10:8080 check inter 2000 rise 2 fall 3
    server app2 10.0.0.11:8080 check inter 2000 rise 2 fall 3
    server app3 10.0.0.12:8080 check inter 2000 rise 2 fall 3

    # Parameters explained:
    # inter 2000  = Check every 2 seconds
    # rise 2      = 2 successful checks to mark healthy
    # fall 3      = 3 failed checks to mark unhealthy

# Advanced: Custom health check
backend database_servers
    option httpchk HEAD /db-health HTTP/1.1\r\nHost:\ localhost

    # Expect specific response
    http-check expect status 200
    http-check expect string "DB_OK"

    server db1 10.0.0.20:3306 check
    server db2 10.0.0.21:3306 check
```

Health Check Endpoint Design

Simple Health Check:

```
python
```

```python
from flask import Flask, jsonify
import psycopg2

app = Flask(__name__)

@app.route('/health')
def health_check():
    """Basic health check - just return 200 OK"""
    return jsonify({"status": "healthy"}), 200

# Load balancer configuration:
# GET /health
# Expected: HTTP 200
# If not 200 or timeout → unhealthy
```

Comprehensive Health Check:

```python
```

```
@app.route('/health')
def comprehensive_health_check():
    """Check all critical dependencies"""

    health_status = {
        "status": "healthy",
        "timestamp": time.time(),
        "checks": {}
    }

    # 1. Check database connectivity
    try:
        db = psycopg2.connect(DB_CONFIG)
        cursor = db.cursor()
        cursor.execute("SELECT 1")
        health_status["checks"]["database"] = "healthy"
        db.close()
    except Exception as e:
        health_status["status"] = "unhealthy"
        health_status["checks"]["database"] = f"unhealthy: {str(e)}"

    # 2. Check Redis cache
    try:
        redis_client.ping()
        health_status["checks"]["redis"] = "healthy"
    except Exception as e:
        health_status["status"] = "unhealthy"
        health_status["checks"]["redis"] = f"unhealthy: {str(e)}"

    # 3. Check disk space
    disk_usage = psutil.disk_usage('/')
    if disk_usage.percent > 90:
        health_status["status"] = "unhealthy"
        health_status["checks"]["disk"] = "unhealthy: > 90% full"
    else:
        health_status["checks"]["disk"] = "healthy"

    # 4. Check memory usage
    memory = psutil.virtual_memory()
    if memory.percent > 95:
        health_status["status"] = "unhealthy"
        health_status["checks"]["memory"] = "unhealthy: > 95% used"
    else:
        health_status["checks"]["memory"] = "healthy"

    # Return appropriate status code
```

```
    status_code = 200 if health_status["status"] == "healthy" else 503

    return jsonify(health_status), status_code

# Example response when healthy:
# {
#   "status": "healthy",
#   "timestamp": 1699564800,
#   "checks": {
#     "database": "healthy",
#     "redis": "healthy",
#     "disk": "healthy",
#     "memory": "healthy"
#   }
# }
```

Readiness vs Liveness Checks:

python

```python
# Kubernetes-style health checks

@app.route('/health/live')
def liveness():
    """
    Liveness check: Is the application running?
    Failure → Restart container
    """
    return jsonify({"status": "alive"}), 200

@app.route('/health/ready')
def readiness():
    """
    Readiness check: Is the application ready to serve traffic?
    Failure → Remove from load balancer (but don't restart)
    """

    # Check if still initializing
    if not app_fully_initialized:
        return jsonify({"status": "not ready"}), 503

    # Check if dependencies available
    if not can_connect_to_database():
        return jsonify({"status": "not ready"}), 503

    if not can_connect_to_cache():
        return jsonify({"status": "not ready"}), 503

    return jsonify({"status": "ready"}), 200
```

Health Check Best Practices

```
1. LIGHTWEIGHT CHECKS
   ✗ Don't: Run complex queries
   ✓ Do: Simple SELECT 1 or ping

2. APPROPRIATE TIMEOUT
   ✗ Don't: 30 second timeout
   ✓ Do: 1-3 second timeout

   Reason: If server is slow to respond to health check,
           it's probably too slow for user requests

3. REASONABLE FREQUENCY
```

✗ Don't: Check every 100ms (too frequent)
✗ Don't: Check every 60s (too slow)
✓ Do: Check every 2-5 seconds

4. FAIL FAST
✗ Don't: Retry health check 10 times
✓ Do: After 2-3 failures, remove from pool

Reason: Quick removal prevents user impact

5. GRADUAL RECOVERY
✗ Don't: One success = back in pool
✓ Do: 2-3 successes = back in pool

Reason: Prevents flapping (in/out/in/out)

6. SEPARATE ENDPOINT
✗ Don't: Use homepage for health check
✓ Do: Dedicated /health endpoint

Reason: Avoid caching, logging noise

Failure Scenarios and Handling

Scenario 1: Server Crashes

Timeline:
00:00 - Server 3 crashes
00:02 - Health check fails (timeout)
00:04 - Health check fails again
00:06 - Health check fails third time
00:06 - Server 3 marked unhealthy
00:06 - Traffic stops going to Server 3

Recovery:
00:10 - Server 3 restarts
00:12 - Health check succeeds
00:14 - Health check succeeds again
00:14 - Server 3 marked healthy
00:14 - Traffic resumes to Server 3

Total downtime for Server 3: 6 seconds
User impact: None (traffic redistributed)

Scenario 2: Database Connection Lost

```python
# Server health check logic
@app.route('/health')
def health():
    try:
        # Try to query database
        db.execute("SELECT 1")
        return "OK", 200
    except DatabaseError:
        # Database down → Report unhealthy
        return "Database connection failed", 503

# Load balancer sees 503 → removes server from pool
# Server attempts to reconnect to database in background
# Once reconnected, health check returns 200
# Load balancer adds server back to pool
```

Scenario 3: Gradual Degradation

Server gradually running out of memory:

Time	Memory	Response Time	Health Check
10:00	60%	50ms	Pass
10:05	70%	80ms	Pass
10:10	85%	150ms	Pass
10:15	92%	500ms	FAIL (timeout)
10:17	94%	Timeout	FAIL
10:19	95%	Timeout	FAIL
10:19	Marked unhealthy → Removed from pool		

Server stops receiving traffic
Memory usage drops (no new requests)
Server recovers
Health checks start passing
Server added back to pool

Circuit Breaker Pattern

Advanced failure detection and handling:

```python
```

```python
from enum import Enum
from datetime import datetime, timedelta

class CircuitState(Enum):
    CLOSED = "closed"      # Normal operation
    OPEN = "open"          # Failing, reject requests
    HALF_OPEN = "half_open"  # Testing if recovered

class CircuitBreaker:
    def __init__(self, failure_threshold=5, timeout=60):
        self.failure_threshold = failure_threshold
        self.timeout = timeout  # seconds
        self.failure_count = 0
        self.last_failure_time = None
        self.state = CircuitState.CLOSED

    def call(self, func):
        """Execute function with circuit breaker protection"""

        # If circuit is OPEN, check if timeout expired
        if self.state == CircuitState.OPEN:
            if datetime.now() - self.last_failure_time > timedelta(seconds=self.timeout):
                self.state = CircuitState.HALF_OPEN
                self.failure_count = 0
            else:
                raise Exception("Circuit breaker is OPEN")

        try:
            # Execute the function
            result = func()

            # Success - reset failure count
            self.failure_count = 0

            # If was HALF_OPEN, close the circuit
            if self.state == CircuitState.HALF_OPEN:
                self.state = CircuitState.CLOSED

            return result

        except Exception as e:
            # Failure - increment count
            self.failure_count += 1
            self.last_failure_time = datetime.now()

            # Open circuit if threshold exceeded
```

```
            if self.failure_count >= self.failure_threshold:
                self.state = CircuitState.OPEN

            raise e

# Usage in load balancer
server_breakers = {
    'server1': CircuitBreaker(),
    'server2': CircuitBreaker(),
    'server3': CircuitBreaker()
}

def route_request(request):
    for server, breaker in server_breakers.items():
        try:
            return breaker.call(lambda: send_to_server(server, request))
        except:
            continue  # Try next server

    raise Exception("All servers unavailable")
```

---

5. Popular Load Balancers

1. NGINX

Type: Layer 7 (HTTP) and Layer 4 (TCP/UDP) **Best for:** Web applications, reverse proxy, API gateway

Basic Configuration:

```
nginx
```

```
# /etc/nginx/nginx.conf

http {
    # Define upstream servers
    upstream web_backend {
        # Load balancing method
        least_conn;  # or: round-robin, ip_hash

        # Server list
        server web1.example.com:8080 weight=3;
        server web2.example.com:8080 weight=2;
        server web3.example.com:8080 weight=1;

        # Backup server (used only if others fail)
        server web4.example.com:8080 backup;
    }

    # Virtual server
    server {
        listen 80;
        server_name example.com;

        # Health check endpoint
        location /health {
            access_log off;
            return 200 "healthy\n";
        }

        # Proxy to backend
        location / {
            proxy_pass http://web_backend;

            # Headers
            proxy_set_header Host $host;
            proxy_set_header X-Real-IP $remote_addr;
            proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;

            # Timeouts
            proxy_connect_timeout 5s;
            proxy_send_timeout 30s;
            proxy_read_timeout 30s;

            # Retry logic
            proxy_next_upstream error timeout http_500 http_502 http_503;
        }
```

```
    }
}
```

Advanced NGINX Features:

```nginx
# SSL/TLS Termination
server {
    listen 443 ssl;
    ssl_certificate /path/to/cert.pem;
    ssl_certificate_key /path/to/key.pem;

    location / {
        proxy_pass http://backend;  # Backend uses HTTP
    }
}

# Caching
proxy_cache_path /var/cache/nginx levels=1:2 keys_zone=my_cache:10m;

server {
    location /api/ {
        proxy_cache my_cache;
        proxy_cache_valid 200 10m;
        proxy_pass http://api_backend;
    }
}

# Rate Limiting
limit_req_zone $binary_remote_addr zone=mylimit:10m rate=10r/s;

server {
    location /api/ {
        limit_req zone=mylimit burst=20;
        proxy_pass http://api_backend;
    }
}
```

---

2. HAProxy

Type: Layer 4 and Layer 7 **Best for:** High-performance, TCP load balancing

Basic Configuration:

```
# /etc/haproxy/haproxy.cfg

global
    maxconn 50000
    log /dev/log local0

defaults
    mode http
    timeout connect 5000ms
    timeout client 50000ms
    timeout server 50000ms

# Frontend (entry point)
frontend http_front
    bind *:80
    default_backend http_back

# Backend (server pool)
backend http_back
    balance roundrobin

    # Health check
    option httpchk GET /health
    http-check expect status 200

    # Servers
    server server1 10.0.0.10:8080 check inter 2000 rise 2 fall 3
    server server2 10.0.0.11:8080 check inter 2000 rise 2 fall 3
    server server3 10.0.0.12:8080 check inter 2000 rise 2 fall 3

# Statistics page
listen stats
    bind *:8404
    stats enable
    stats uri /stats
    stats refresh 30s
    stats admin if TRUE
```

Advanced HAProxy:

```
# ACL-based routing
frontend web
    bind *:80

    # Define ACLs
    acl is_api path_beg /api
    acl is_static path_beg /static
    acl is_admin hdr(host) -i admin.example.com

    # Route based on ACLs
    use_backend api_servers if is_api
    use_backend static_servers if is_static
    use_backend admin_servers if is_admin
    default_backend web_servers

# Stick tables (session persistence)
backend app_servers
    balance roundrobin
    stick-table type ip size 1m expire 30m
    stick on src  # Stick based on source IP

    server app1 10.0.0.20:8080 check
    server app2 10.0.0.21:8080 check
```

---

3. AWS Elastic Load Balancer (ELB)

Types:

1. **Application Load Balancer (ALB)** - Layer 7, HTTP/HTTPS
2. **Network Load Balancer (NLB)** - Layer 4, TCP/UDP
3. **Classic Load Balancer (CLB)** - Legacy

ALB Configuration (Terraform):

```
hcl
```

```
# Application Load Balancer
resource "aws_lb" "web" {
  name               = "web-alb"
  internal           = false
  load_balancer_type = "application"
  security_groups    = [aws_security_group.lb_sg.id]
  subnets            = aws_subnet.public.*.id
}

# Target Group
resource "aws_lb_target_group" "web" {
  name     = "web-targets"
  port     = 80
  protocol = "HTTP"
  vpc_id   = aws_vpc.main.id

  # Health check
  health_check {
    enabled             = true
    healthy_threshold   = 2
    interval            = 30
    matcher             = "200"
    path                = "/health"
    port                = "traffic-port"
    timeout             = 5
    unhealthy_threshold = 3
  }
}

# Listener
resource "aws_lb_listener" "web" {
  load_balancer_arn = aws_lb.web.arn
  port              = "80"
  protocol          = "HTTP"

  default_action {
    type             = "forward"
    target_group_arn = aws_lb_target_group.web.arn
  }
}

# Register targets
resource "aws_lb_target_group_attachment" "web" {
  count            = 3
  target_group_arn = aws_lb_target_group.web.arn
  target_id        = aws_instance.web[count.index].id
```

```
  port            = 80
}
```

ALB Features:

- Path-based routing
- Host-based routing
- Lambda target support
- WebSocket support
- HTTP/2 support

Load Balancer Comparison

Feature	NGINX	HAProxy	AWS ALB	AWS NLB
Layer	4 & 7	4 & 7	7	4
Performance	High	Very High	High	Very High
Cost	Free	Free	Pay/hour	Pay/hour
Management	Manual	Manual	Managed	Managed
SSL Term	Yes	Yes	Yes	Yes
Auto-scale	No	No	Yes	Yes
Best for	Web/API	TCP/HTTP	HTTP/S	TCP/UDP

Throughput Comparison:
NGINX: ~50,000 req/s (single instance)
HAProxy: ~100,000 req/s (single instance)
AWS ALB: ~500 req/s per LCU (auto-scales)
AWS NLB: Millions of req/s (auto-scales)

Key Takeaways

1. **Load Balancing Purpose:**
   - Distribute traffic across servers
   - Eliminate single point of failure
   - Enable horizontal scaling
   - Improve performance and reliability
2. **Algorithms:**

- Round Robin: Simple, fair distribution
- Least Connections: Smart for varying workloads
- IP Hash: Session persistence
- Choose based on use case

3. **Layer 4 vs 7:**
   - Layer 4: Fast, simple, any protocol
   - Layer 7: Intelligent routing, HTTP-aware
   - Often use both in tandem
4. **Health Checks:**
   - Critical for reliability
   - Active checks detect problems early
   - Design lightweight health endpoints
   - Configure appropriate thresholds
5. **Tools:**
   - NGINX: Great for web/API, feature-rich
   - HAProxy: High performance, TCP-focused
   - AWS ELB: Managed, auto-scaling, cloud-native

Practice Questions

1. Your app has 10 servers. Design a health check strategy that minimizes user impact when a server fails.
2. When would you choose Layer 4 over Layer 7 load balancing?
3. Design a load balancing solution for a video streaming service with users worldwide.

Next Chapter Preview

In Chapter 5, we'll explore **Caching**:

- Why caching dramatically improves performance
- Caching strategies (cache-aside, write-through, etc.)
- Cache eviction policies (LRU, LFU)
- Distributed caching with Redis
- CDN for static content

Ready to continue?