Load Balancing

Duration: 60 minutes **Level:** Intermediate

Session Agenda

- ☐ Introduction to Load Balancing (10 min)
- ☐ Types of Load Balancers (10 min)
- ☐ Load Balancing Algorithms (15 min)
- ☐ Implementation & Tools (15 min)
- ☐ Best Practices & Troubleshooting (10 min)

Learning Objectives

By the end of this session, you will understand: - What load balancing is and why it matters - Different types of load balancers - Common load balancing algorithms - How to implement basic load balancing - Best practices for production environments

---

1. Introduction to Load Balancing (10 min)

What is Load Balancing?

Load Balancing is the process of distributing network traffic across multiple servers to ensure: - High availability - Scalability - Reliability - Performance optimization

Why Do We Need Load Balancing?

> [!info] Key Problem A single server has limited resources (CPU, memory, network bandwidth). As traffic grows, a single server becomes a bottleneck and single point of failure.

Benefits: 1. **Prevents server overload** - Distributes requests evenly 2. **Increases availability** - If one server fails, others continue 3. **Enables horizontal scaling** - Add more servers as needed 4. **Reduces latency** - Routes users to nearest/fastest server 5. **Enables maintenance** - Take servers offline without downtime

Real-World Analogy

Think of a grocery store with multiple checkout lanes: - **Without load balancing:** Everyone queues at one register - **With load balancing:** Customers distributed across multiple registers

---

2. Types of Load Balancers (10 min)

By Implementation Layer

Layer 4 (Transport Layer) Load Balancing

- Operates at TCP/UDP level
- Routes based on IP address and port
- Fast but limited routing decisions
- Cannot inspect packet content

```
Client Request → Load Balancer (checks IP:Port) → Backend Server
                        ↓
                Forwards packets
                (no content inspection)
```

Example: AWS Network Load Balancer (NLB)

Layer 7 (Application Layer) Load Balancing

- Operates at HTTP/HTTPS level
- Routes based on content (URL, headers, cookies)
- Slower but more intelligent routing
- Can perform SSL termination

```
Client Request → Load Balancer (checks URL/headers) → Backend Server
                        ↓
          Can route /api/* to API servers
          Can route /static/* to CDN
```

Example: AWS Application Load Balancer (ALB), Nginx

Metadata Available at Each Layer for Load Balancing

Layer 3 (Network) - GWLB

Metadata	Example
Source IP	`192.168.1.100`
Destination IP	`10.0.0.50`
Protocol	TCP, UDP, ICMP

Layer 4 (Transport) - NLB

Metadata	Example
All L3 metadata	—
Source Port	`52431`
Destination Port	`443`, `3306`
TCP Flags	SYN, ACK, FIN

Layer 7 (Application) - ALB

Metadata	Example
All L3 + L4 metadata	—
HTTP Method	`GET`, `POST`, `DELETE`
URL Path	`/api/v1/users`
Query String	`?id=123&sort=name`
Host Header	`api.example.com`
Headers	`User-Agent`, `Cookie`, `Authorization`
Body/Content	JSON, Form data
SNI (TLS)	`www.example.com`

By Architecture

Type	Description	Use Case
Hardware	Physical appliances (F5, Citrix)	High-performance enterprise
Software	Software-based (Nginx, HAProxy)	Flexible, cost-effective
DNS-based	DNS round-robin	Simple geographic distribution
Cloud-based	Managed services (AWS ELB, Azure LB)	Cloud-native applications

---

3. Load Balancing Algorithms (15 min)

1. Round Robin

How it works: Distributes requests sequentially across servers.

```
Request 1 → Server A
Request 2 → Server B
Request 3 → Server C
Request 4 → Server A (cycle repeats)
```

Pros: - Simple and fair distribution - Easy to implement

Cons: - Doesn't consider server load or capacity - All servers must have equal capacity

2. Weighted Round Robin

How it works: Assigns weights based on server capacity.

```
Servers:
  - Server A (weight: 3) → Gets 3 requests
  - Server B (weight: 2) → Gets 2 requests
  - Server C (weight: 1) → Gets 1 request
```

Use case: When servers have different specifications.

3. Least Connections

How it works: Routes to server with fewest active connections.

```
Server A: 45 connections
Server B: 32 connections ← New request goes here
Server C: 50 connections
```

Pros: - Better for varying request durations - More intelligent than round robin

Cons: - Requires tracking connection state

4. Least Response Time

How it works: Routes to server with fastest response time.

```
# Pseudocode
def select_server(servers):
    metrics = []
    for server in servers:
        avg_response = server.average_response_time
        active_connections = server.connections
        score = avg_response * active_connections
```

```
        metrics.append((score, server))

    return min(metrics)[1]  # Return server with lowest score
```

5. IP Hash / Session Persistence

How it works: Uses client IP to consistently route to same server.

```
hash(client_ip) % num_servers = server_index
```

```
Example:
Client 192.168.1.100 → hash → always routes to Server B
```

Use case: When session data is stored locally on servers.

> [!warning] Sticky Sessions While useful, sticky sessions can cause uneven load distribution and make scaling harder. Consider centralized session storage instead.

6. Least Bandwidth

Routes to server currently serving least amount of traffic (Mbps).

Comparison Table

Algorithm	Complexity	Session Aware	Best For
Round Robin	Low	No	Equal servers, stateless apps
Weighted RR	Low	No	Servers with different capacity
Least Connections	Medium	Yes	Variable request duration
Least Response Time	High	Yes	Performance-critical apps
IP Hash	Medium	Yes	Session persistence needed

---

4. Implementation & Tools (15 min)

Popular Load Balancing Tools

Nginx Configuration Example:

```
http {
    upstream backend {
        # Load balancing method
        least_conn;  # or: ip_hash, hash, random

        # Backend servers
        server backend1.example.com weight=3;
        server backend2.example.com weight=2;
        server backend3.example.com weight=1 backup;

        # Health checks
        server backend4.example.com max_fails=3 fail_timeout=30s;
    }

    server {
        listen 80;
```

```
        location / {
            proxy_pass http://backend;
            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 10s;
            proxy_read_timeout 10s;
        }
    }
}
```

HAProxy Configuration Example:

```
global
    log /dev/log local0
    maxconn 4096

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

frontend http_front
    bind *:80
    default_backend http_back

backend http_back
    balance roundrobin
    option httpchk GET /health

    server server1 192.168.1.10:8080 check weight 3
    server server2 192.168.1.11:8080 check weight 2
    server server3 192.168.1.12:8080 check weight 1 backup
```

Cloud Load Balancers

```
resource "aws_lb" "main" {
  name               = "app-load-balancer"
  internal           = false
  load_balancer_type = "application"
  security_groups    = [aws_security_group.lb_sg.id]
  subnets           = aws_subnet.public[*].id

  enable_deletion_protection = true
}

resource "aws_lb_target_group" "app" {
  name     = "app-target-group"
```

```
  port     = 80
  protocol = "HTTP"
  vpc_id   = aws_vpc.main.id

  health_check {
    enabled             = true
    healthy_threshold   = 2
    interval            = 30
    matcher             = "200"
    path                = "/health"
    timeout             = 5
    unhealthy_threshold = 2
  }
}

resource "aws_lb_listener" "front_end" {
  load_balancer_arn = aws_lb.main.arn
  port              = "443"
  protocol          = "HTTPS"
  ssl_policy        = "ELBSecurityPolicy-2016-08"
  certificate_arn   = aws_acm_certificate.cert.arn

  default_action {
    type             = "forward"
    target_group_arn = aws_lb_target_group.app.arn
  }
}
```

AWS Application Load Balancer (Terraform)

Simple Python Load Balancer Example

```
from flask import Flask, request
import requests
import itertools

app = Flask(__name__)

# Backend servers
SERVERS = [
    "http://server1:5000",
    "http://server2:5000",
    "http://server3:5000"
]

# Round-robin iterator
server_pool = itertools.cycle(SERVERS)

@app.route('/', defaults={'path': ''})
@app.route('/<path:path>', methods=['GET', 'POST', 'PUT', 'DELETE'])
def load_balance(path):
    # Get next server
    server = next(server_pool)
    url = f"{server}/{path}"
```

```
    # Forward request
    try:
        response = requests.request(
            method=request.method,
            url=url,
            headers={k:v for k,v in request.headers if k != 'Host'},
            data=request.get_data(),
            cookies=request.cookies,
            allow_redirects=False,
            timeout=5
        )
        return response.content, response.status_code
    except requests.exceptions.RequestException as e:
        return f"Server error: {e}", 503

if __name__ == '__main__':
    app.run(host='0.0.0.0', port=8080)
```

---

5. Best Practices & Troubleshooting (10 min)

Best Practices

1. Health Checks Always configure health checks to automatically remove unhealthy servers.

```
health_check:
  endpoint: /health
  interval: 10s
  timeout: 5s
  healthy_threshold: 2
  unhealthy_threshold: 3
```

2. Connection Draining Allow existing connections to complete before removing server.

```
# Nginx
upstream backend {
    server backend1.example.com slow_start=30s;
}
```

3. SSL/TLS Termination Terminate SSL at load balancer to reduce backend server load.

```
Client (HTTPS) → Load Balancer (SSL termination) → Backend (HTTP)
```

4. Monitoring & Logging **Key Metrics to Monitor:** - Request rate (requests/second) - Response time (p50, p95, p99) - Error rate (4xx, 5xx) - Backend server health - Connection pool utilization

```
# Kubernetes HPA example
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: app-hpa
spec:
```

```yaml
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: app
  minReplicas: 3
  maxReplicas: 10
  metrics:
  - type: Resource
    resource:
      name: cpu
      target:
        type: Utilization
        averageUtilization: 70
```

5. Auto-Scaling Integration

Common Issues & Solutions

Issue	Symptoms	Solution
Uneven Distribution	Some servers overloaded	Use least_conn or weighted algorithms
Session Loss	Users logged out randomly	Implement sticky sessions or centralized sessions
Health Check Failures	False positives	Tune health check thresholds and intervals
SSL Certificate Issues	HTTPS errors	Ensure certificates valid and properly configured
Timeout Errors	504 Gateway Timeout	Increase timeout values, check backend performance

Troubleshooting Checklist

- ☐ Check load balancer logs
- ☐ Verify backend server health
- ☐ Test connectivity to each backend
- ☐ Review load balancing algorithm choice
- ☐ Check SSL/TLS configuration
- ☐ Verify security group/firewall rules
- ☐ Monitor resource utilization
- ☐ Test failover scenarios

Architecture Patterns

Pattern 1: Simple Web Application

```
Internet       Load Balancer
```

```
    Web             Web             Web
    Server 1        Server 2        Server 3


                        Database
```

Pattern 2: Microservices Architecture

```
Internet        External LB


      API             Auth            Static
    Gateway           Service         Content


                Internal LB


    Service         Service         Service
       A               B               C
```

Additional Resources

Documentation

- [[Nginx Load Balancing Guide]]
- [[HAProxy Documentation]]
- [[AWS ELB Best Practices]]

Related Topics

- #networking
- #scalability
- #high-availability
- #devops

Books & Articles

- "The Art of Scalability" by Martin L. Abbott
- "Designing Data-Intensive Applications" by Martin Kleppmann

---

Key Takeaways

1. **Load balancing is essential** for scalable, highly-available applications
2. **Choose the right algorithm** based on your application needs:
   - Stateless apps → Round Robin
   - Variable request times → Least Connections
   - Session-heavy apps → IP Hash (but consider alternatives)
3. **Always implement health checks** to detect and remove unhealthy servers
4. **Monitor key metrics** to ensure optimal performance
5. **Layer 4 vs Layer 7**: Trade-off between speed and intelligence

---

Practical Exercise

Scenario: You have a web application with: - 3 backend servers (different capacities) - Users must maintain login sessions - Some requests take longer than others - Need zero-downtime deployments

Question: Which load balancing strategy would you use and why?

Suggested Solution: 1. Use **Weighted Least Connections** algorithm 2. Implement **centralized session storage** (Redis/Memcached) 3. Configure **health checks** with proper thresholds 4. Enable **connection draining** for deployments 5. Set up **monitoring** for response times and error rates

---

Q&A Session

Time remaining for questions and discussion

Common Questions: 1. **When should I use Layer 4 vs Layer 7?** - Use Layer 4 for pure TCP/UDP traffic, high performance needs - Use Layer 7 for HTTP/HTTPS with content-based routing

2. **How many load balancers do I need?**
   - At minimum 2 for redundancy (active-passive or active-active)
   - Cloud providers handle this automatically
3. **What about load balancing databases?**
   - Different approach: read replicas, connection pooling
   - Tools: ProxySQL, PgBouncer, HAProxy

Notes Section

Use this space for your own notes during the session

Session End - Thank You!

#load-balancing #infrastructure #session-notes