# Load Balancer Algorithms: Concepts, Calculations, and Configurations

Load balancers are the backbone of modern distributed systems, ensuring that traffic is efficiently distributed across servers to prevent overloads and maximize performance. This article provides a detailed explanation of eight commonly used load-balancing algorithms, including their mechanics, calculation methods, and practical configuration examples for different tools and platforms.

---

## 1. Round Robin

### Concept

Round Robin distributes requests cyclically across all available servers. Each server receives an equal number of requests, regardless of its capacity or performance.

### Example Calculation

Consider three servers (S1, S2, S3):

- Request 1 → S1
- Request 2 → S2
- Request 3 → S3
- Request 4 → S1 (cycle repeats)

#### Assumptions

- Servers: S1, S2, S3
- Incoming requests: 9
  Each server will handle 3 requests.

### Configuration

#### NGINX

```
upstream my_backend {
   server 192.168.1.1;
   server 192.168.1.2;
   server 192.168.1.3;
}
```

### HAProxy

```
backend my_backend
    balance roundrobin
    server s1 192.168.1.1:80 check
    server s2 192.168.1.2:80 check
    server s3 192.168.1.3:80 check
```

### Best Use Case

Simple systems with uniform server capabilities.

---

## 2. Least Connections

### Concept

Routes requests to the server with the fewest active connections. Ideal for workloads with varying request durations.

### Example Calculation

Active connections:

- S1: 5
- S2: 2
- S3: 3
  The next request will go to **S2** because it has the fewest active connections.

### Assumptions

- Servers dynamically handle different request loads.
- Server S2 can accept a new connection faster than others.

### Configuration

### NGINX

```
upstream my_backend {
    least_conn;
    server 192.168.1.1;
    server 192.168.1.2;
    server 192.168.1.3;
}
```

### HAProxy

```
backend my_backend
    balance leastconn
    server s1 192.168.1.1:80 check
    server s2 192.168.1.2:80 check
    server s3 192.168.1.3:80 check
```

### Best Use Case

Dynamic workloads where connection durations vary significantly.

---

## 3. Weighted Round Robin

### Concept

Each server is assigned a weight. Servers with higher weights handle more requests proportionally.

### Example Calculation

Weights:

- S1: 2
- S2: 1
  Request distribution for 6 incoming requests:
- S1: 4 requests (2/3 of total requests)
- S2: 2 requests (1/3 of total requests)

### Configuration

### HAProxy

```
backend my_backend
    balance roundrobin
    server s1 192.168.1.1:80 weight 2 check
    server s2 192.168.1.2:80 weight 1 check
```

### Best Use Case

Systems with heterogeneous servers having varying capacities.

---

# 4. IP Hash

## Concept

Routes requests from a specific client to a specific server using a hash of the client's IP address. Ensures session persistence.

## Example Calculation

Client IP: 192.168.1.100
Hash calculation:
hash(192 + 168 + 1 + 100) % 3 = 2
Request routed to S3.

## Configuration

### NGINX

```
upstream my_backend {
    ip_hash;
    server 192.168.1.1;
    server 192.168.1.2;
    server 192.168.1.3;
}
```

## Best Use Case

Systems requiring session persistence without additional mechanisms like cookies.

---

# 5. Random

## Concept

Assigns requests randomly to servers. Simplicity is its strength.

## Example Calculation

Servers: S1, S2, S3
Random function generates:

- Request 1 → S3
- Request 2 → S1
- Request 3 → S2

### Configuration

#### NGINX with Lua

```
upstream my_backend {
    server 192.168.1.1;
    server 192.168.1.2;
    server 192.168.1.3;
}
```

### Best Use Case

Basic setups where server performance is uniform.

---

## 6. Least Response Time

### Concept

Routes requests to the server with the lowest response time. It continuously monitors and updates response time data for each server.

### Example Calculation

Servers:

- S1: 120ms
- S2: 80ms
- S3: 100ms
  Next request routed to **S2**.

### Configuration

#### AWS Application Load Balancer

AWS ALB uses built-in monitoring for least response time but doesn't require explicit configuration.

#### Custom Implementation

In custom systems, you can integrate health checks that monitor response times and dynamically adjust traffic routing.

### Best Use Case

Real-time applications requiring low latency.

---

# 7. Geographic (Geo) Routing

## Concept

Routes users to the nearest server based on their geographic location.

## Example Calculation

Client location: **New York, USA**
Available servers:

- US-East (Virginia)
- US-West (Oregon)
  Request routed to **US-East (Virginia)** as it's closer.

## Configuration

### AWS Route 53

```
{
  "Type": "AWS::Route53::RecordSet",
  "Properties": {
    "GeoLocation": {
      "CountryCode": "US"
    },
    "ResourceRecords": ["192.168.1.1"],
    "Type": "A",
    "TTL": "60"
  }
}
```

## Best Use Case

Content delivery networks (CDNs) or region-specific services.

---

# 8. Consistent Hashing

## Concept

Distributes requests based on a hash function, ensuring the same client is routed to the same server unless the server fails.

## Example Calculation

Client ID: 12345
Hash calculation:
hash(12345) % 3 = 0
Request routed to S1.

## Configuration

### NGINX

```
upstream my_backend {
    hash $request_uri consistent;
    server 192.168.1.1;
    server 192.168.1.2;
    server 192.168.1.3;
}
```

## Best Use Case

Caching systems or services requiring sticky sessions.

---

## Key Takeaways

- **Round Robin** and **Random** are best for simple, uniform systems.
- **Least Connections** and **Least Response Time** are ideal for dynamic workloads.
- **Weighted Round Robin** supports heterogeneous server environments.
- **IP Hash** and **Consistent Hashing** ensure session persistence.
- **Geo Routing** optimizes traffic based on user location.

By understanding the nuances of each algorithm and configuring them appropriately, you can maximize the efficiency and reliability of your load-balancing strategy. Always monitor and adapt configurations as workloads evolve!