# Load Balancing Algorithms

## 🔁 1. Round Robin (RR)

- **Working**: Distributes requests sequentially to each server in a circular order.
- **Example**: Server1 → Server2 → Server3 → Server1 ...
- **Pros**:
  - Simple and fair.
  - No need for server state info.
- **Cons**:
  - Doesn't consider server load or capacity.

```
class RoundRobin:
    def __init__(self, servers):
        self.servers = servers
        self.current_index = -1

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

---

## ⚖️ 2. Weighted Round Robin (WRR)

- **Working**: Similar to Round Robin but each server is assigned a weight. Higher-weight servers receive more requests.
- **Example**: Weights [5, 1, 1] → Server1 gets 5x more requests.
- **Pros**:
  - Better distribution based on capacity.
- **Cons**:
  - Slightly complex.
  - Not dynamic to real-time load.

```
class WeightedRoundRobin:
    def __init__(self, servers, weights):
        self.servers = servers
        self.weights = weights
        self.current_index = -1
        self.current_weight = 0

    def get_next_server(self):
        while True:
            self.current_index = (self.current_index + 1) % len(self.servers)
            if self.current_index == 0:
                self.current_weight -= 1
```

```
                if self.current_weight <= 0:
                    self.current_weight = max(self.weights)
            if self.weights[self.current_index] >= self.current_weight:
                return self.servers[self.current_index]
```

---

## 📉 3. Least Connections

- **Working**: Routes the request to the server with the **fewest active connections**.
- **Use Case**: Good when sessions are long-lived (e.g., database connections).
- **Pros**:
  - Dynamically adapts to server load.
- **Cons**:
  - Requires tracking active connections.

```
class LeastConnections:
    def __init__(self, servers):
        self.servers = {server: 0 for server in servers}

    def get_next_server(self):
        min_connections = min(self.servers.values())
        least_loaded_servers = [server for server, connections in
self.servers.items() if connections == min_connections]
        selected_server = random.choice(least_loaded_servers)
        self.servers[selected_server] += 1
        return selected_server

    def release_connection(self, server):
        if self.servers[server] > 0:
            self.servers[server] -= 1
```

---

## ⏱️ 4. Least Response Time

- **Working**: Sends the request to the server with the **lowest average response time**.
- **Pros**:
  - Highly dynamic and responsive to performance.
- **Cons**:
  - Needs constant monitoring and timing.

```python
class LeastResponseTime:
    def __init__(self, servers):
        self.servers = servers
        self.response_times = [0] * len(servers)

    def get_next_server(self):
        min_response_time = min(self.response_times)
        min_index = self.response_times.index(min_response_time)
        return self.servers[min_index]

    def update_response_time(self, server, response_time):
        index = self.servers.index(server)
        self.response_times[index] = response_time
```

---

## 🎲 5. Random

- **Working:** Assigns each request to a random server.
- **Pros:**
  - Simple to implement.
- **Cons:**
  - May cause uneven distribution.

```python
def simulate_response_time():
    delay = random.uniform(0.1, 1.0)
    time.sleep(delay)
    return delay
```

---

## 📦 Summary Table

| Algorithm | Adaptive | Tracks Load | Use Case |
|---|---|---|---|
| Round Robin | ❌ | ❌ | Uniform servers |
| Weighted RR | ❌ | ❌ | Servers with different capacity |
| Least Connections | ✅ | ✅ | Varying session lengths |
| Least Response Time | ✅ | ✅ | Performance-sensitive systems |
| Random | ❌ | ❌ | Lightweight or dev setups |

---

## Demonstration

```
def demonstrate_algorithm(algorithm_name, load_balancer, iterations=6,
use_response_time=False, use_connections=False):
    print(f"\n---- {algorithm_name} ----")

    for i in range(iterations):
        server = load_balancer.get_next_server()
        print(f"Request {i + 1} → {server}")

        if use_response_time:
            response_time = simulate_response_time()
            load_balancer.update_response_time(server, response_time)
            print(f"Response Time: {response_time:.2f}s")

        if use_connections:
            load_balancer.release_connection(server)


if __name__ == "__main__":
    servers = ["Server1", "Server2", "Server3"]

    # Round Robin
    rr = RoundRobin(servers)
    demonstrate_algorithm("Round Robin", rr)

    # Weighted Round Robin
    weights = [5, 1, 1]
    wrr = WeightedRoundRobin(servers, weights)
    demonstrate_algorithm("Weighted Round Robin", wrr, iterations=7)

    # Least Connections
    lc = LeastConnections(servers)
    demonstrate_algorithm("Least Connections", lc, use_connections=True)

    # Least Response Time
    lrt = LeastResponseTime(servers)
    demonstrate_algorithm("Least Response Time", lrt, use_response_time=True)
```