Troubleshooting and Resolving Load Balancer Traffic Imbalance: A Deep Dive into Diagnostics, Logs, and Performance Optimization

To troubleshoot and resolve this uneven load distribution issue, we need to systematically investigate multiple areas: load balancer configuration, server health, CPU utilization patterns, request distribution, and system logs. Below is a step-by-step approach with detailed technical explanations, commands, logs, and recommendations.

Step 1: Identify the Overloaded Server and Analyze Traffic Distribution

First, we need to determine which specific server is experiencing 100% CPU utilization and whether the load balancer is evenly distributing traffic.

1.1 Check CPU Usage Across All Servers

Run the following command on each backend server to check CPU utilization:

```
top -b -n1 | grep "Cpu(s)"

OR

mpstat -P ALL 1 5
```

Example Output:

```
CPU   %usr   %nice %sys %iowait %irq %soft %steal %guest %idle

all     96.5   0.0  3.5   0.0  0.0  0.0  0.0   0.0  0.0

0       99.9   0.0  0.1   0.0  0.0  0.0  0.0   0.0  0.0 ← Overloaded core

1       30.0   0.0  5.0   0.0  0.0  0.0  0.0   0.0 65.0
```

🚨 If one server consistently shows **100% CPU**, while others remain under **50%**, it indicates an **uneven traffic distribution issue.**

1.2 Verify Load Balancer Traffic Distribution

For AWS ALB/NLB

Check load balancer request distribution to backend servers:

```
aws elb describe-load-balancers --query "LoadBalancerDescriptions[*].Instances"

aws elb describe-target-health --target-group-arn <TARGET_GROUP_ARN>
```

For logs:

```
cat /var/log/httpd/access.log | awk '{print $1}' | sort | uniq -c | sort -nr
```

This command will show which IPs are receiving the most traffic.

Example Output:

```
Server-1 (100% CPU) --> Received 80% of total traffic

Server-2 (50% CPU)  --> Received 10% of total traffic

Server-3 (50% CPU)  --> Received 10% of total traffic
```

🚨 If one server is receiving **disproportionate requests**, there may be an issue with **stickiness, load balancer health checks, or session affinity.**

Step 2: Investigate Load Balancer Configuration Issues

2.1 Check Load Balancer Algorithm

Ensure the load balancer is using a **round-robin or least connections** method.

For AWS ALB:

Check the ALB routing algorithm:

```
aws elbv2 describe-target-groups --query "TargetGroups[*].LoadBalancerArns"
```

If using **sticky sessions**, it could be a problem:

```
aws elbv2 describe-target-groups --query "TargetGroups[*].StickinessConfig"
```

🚨 **Issue:** If session stickiness is enabled with a long TTL, users might be pinned to an overloaded server.

For Nginx Load Balancer:

Check /etc/nginx/nginx.conf:

```
upstream backend {

  server app1.example.com weight=1;

  server app2.example.com weight=1;

  server app3.example.com weight=1;

}
```

🚨 **Issue:** If weights are imbalanced, traffic might not be evenly distributed.

For HAProxy Load Balancer:

Check /etc/haproxy/haproxy.cfg:

```
backend servers
  balance roundrobin
  server server1 10.0.0.1:80 check
  server server2 10.0.0.2:80 check
  server server3 10.0.0.3:80 check
```

🚨 **Issue:** If balance is set to source, it could cause an imbalance.

Step 3: Verify Application-Specific Issues

If load balancing is **correctly configured**, the issue might be within the application itself.

3.1 Check Active Connections per Server

On the overloaded server, check how many connections are open:

```
netstat -an | grep :80 | wc -l
```

OR

```
ss -s
```

Example Output:

```
Total: 5000 active connections on Server-1 (Overloaded)
```

🚨 **If this number is significantly higher than other servers**, the application might have **long-running requests or inefficient request handling.**

3.2 Check Slow Running Requests

Analyze the slowest requests in the access logs:

```
cat /var/log/nginx/access.log | awk '{print $NF, $7, $9}' | sort -nr | head -10
```

🚨 If a specific request (e.g., /api/report) is causing long execution times, it might be a **CPU-intensive process** that needs optimization.

Step 4: Investigate Memory Leaks, Thread Bottlenecks, and GC Issues

4.1 Analyze Java Thread Dump (If Java Application)

Capture a thread dump:

```
jstack -l <PID> > thread_dump.txt
```

🚨 Look for **thread contention** or excessive **CPU-consuming threads**.

4.2 Check Java Garbage Collection (GC) Performance

Enable GC logs and analyze:

```
jstat -gcutil <PID> 1000 5
```

Example:

```
 S0    S1    E     O     M    CCS   YGC  YGCT  FGC  FGCT   GCT
 0.0  12.5  98.3  85.7  67.1  55.2  1002  23.45  15  10.78  34.23
```

🚨 **High Full GC time (>10%) could indicate memory leaks or inefficient object allocation.**

Step 5: Recommendations & Fixes

Based on the findings, implement the following fixes:

1. Adjust Load Balancer Configuration

- **Disable Sticky Sessions:** Unless absolutely required.
- **Change Load Balancer Algorithm:** Use **Least Connections** instead of Round Robin.
- **Reconfigure Health Checks:** Ensure all backend servers are **healthy** to prevent LB overloading one instance.

2. Optimize Server Performance

- **Enable Auto Scaling:** To spin up new instances when traffic spikes.
- **Tune JVM Parameters (If Java-based):**
- -XX:+UseG1GC -Xms2g -Xmx4g -XX:+HeapDumpOnOutOfMemoryError
- **Optimize Application Code:** Identify **long-running DB queries, memory leaks, or inefficient CPU-bound processes.**

3. Reduce Connection Load

- **Implement Connection Pooling:** If too many connections are open, use:

  ```
  jdbc:mysql://host/db?useSSL=false&serverTimezone=UTC&useLegacyDatetimeCode=false&rewriteBatchedStatements=true
  ```

- **Limit Request Rate:** Add **Rate Limiting (NGINX Example)**:

  ```
  limit_req_zone $binary_remote_addr zone=one:10m rate=10r/s;
  ```

4. Enable Auto-Recovery

- **Enable Horizontal Auto Scaling (AWS ECS/K8s):**

  ```
  kubectl autoscale deployment my-app --cpu-percent=70 --min=2 --max=10
  ```

- **Use AWS ALB Target Tracking Policy:**

  ```
  aws application-autoscaling put-scaling-policy --policy-type TargetTrackingScaling \
    --resource-id service/my-app \
    --target-value 50 \
    --scalable-dimension ecs:service:DesiredCount
  ```

Conclusion

By following the above steps, you can confirm and fix load balancing issues using logs, metrics, and performance tuning strategies to ensure even traffic distribution and stable system performance.