Author: Madhurima Rawat

Cloud Load Balancing and Auto Scaling

The experiment configures load balancing and auto-scaling using AWS CLI, LocalStack, Docker, and Nginx. It simulates cloud traffic distribution across instances for efficient workload management. The setup demonstrates scalable deployment in a local cloud environment.

What is Load Balancing?

Load balancing is the process of distributing incoming network traffic across multiple servers to ensure no single server bears too much demand. This improves **performance, reliability, and fault tolerance** by preventing overload and ensuring continuous availability.

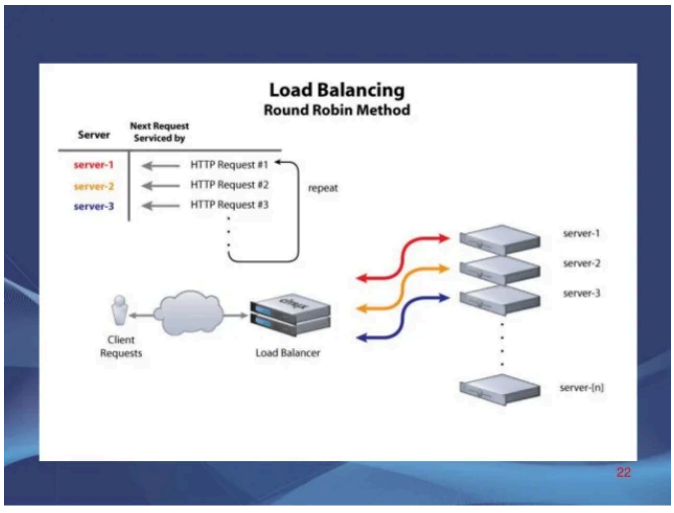
Key Benefits of Load Balancing:

- **Improved Performance:** Spreads traffic efficiently across multiple servers.
- **High Availability & Redundancy:** Ensures uptime even if a server fails.
- **Scalability:** Easily accommodates increased traffic by adding more servers.
- **Optimized Resource Utilization:** Ensures servers are used efficiently.

Types of Load Balancing:

1. **Round Robin:** Distributes requests sequentially across servers.
2. **Least Connections:** Directs traffic to the server with the fewest connections.
3. **IP Hash:** Uses the client's IP address to route requests consistently to the same server.
4. **Weighted Load Balancing:** Assigns different weights to servers based on their capacity.

Load Balancing Illustration



This image visually represents **load balancing**, showing how traffic is distributed among multiple servers to ensure high availability and reliability.

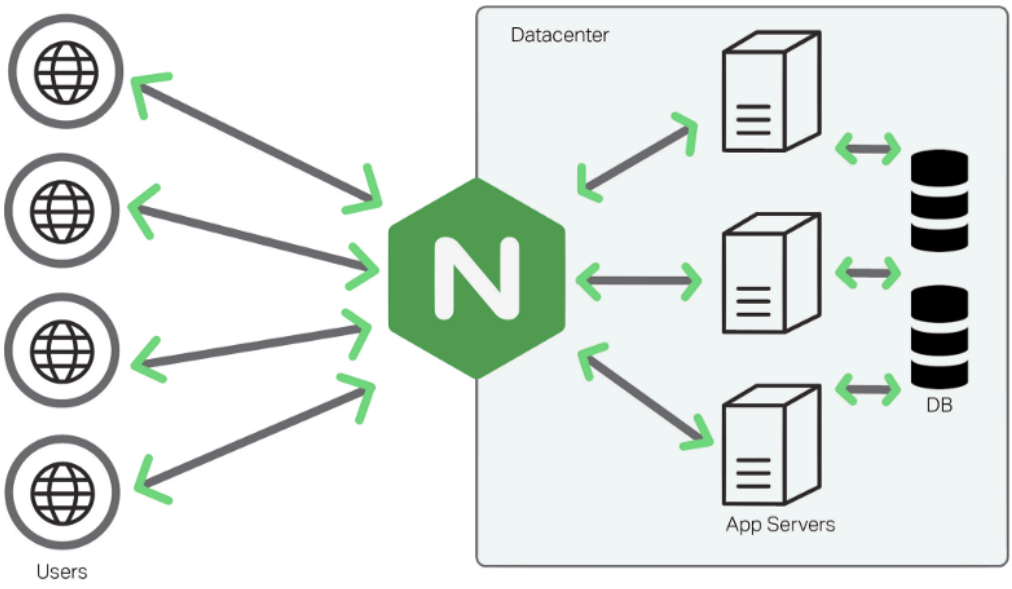
What is Nginx?

Nginx (pronounced "engine-x") is a **high-performance web server and reverse proxy** designed for speed, stability, and scalability. It is widely used for **load balancing, caching, and serving static content** efficiently.

Why Use Nginx?

- **Reverse Proxy & Load Balancing:** Distributes requests across multiple backend servers.
- **High Performance:** Handles thousands of concurrent connections efficiently.
- **Security Features:** Protects against DDoS attacks and unauthorized access.
- **Efficient Static Content Handling:** Quickly serves HTML, CSS, JavaScript, and images.
- **Support for Microservices:** Works seamlessly with modern containerized applications.

Nginx Overview



This image provides a high-level view of Nginx's role in handling requests, acting as a **reverse proxy, load balancer, and web server** for modern applications.

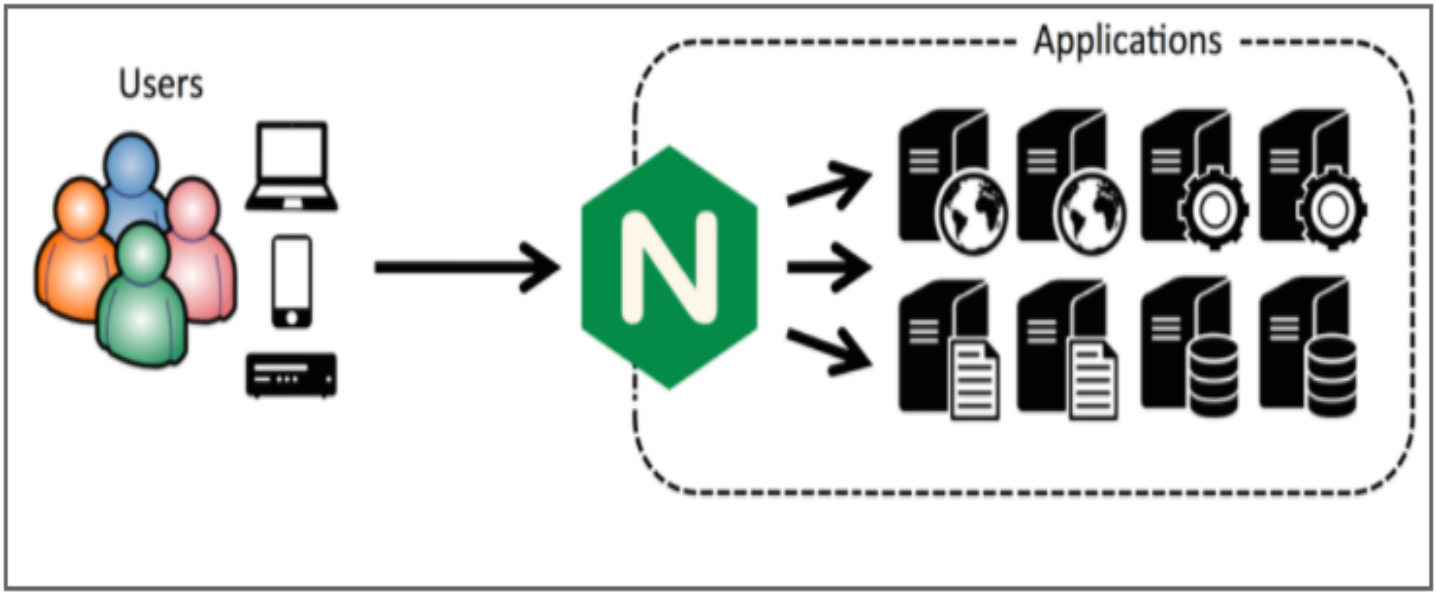
How Nginx Works as a Load Balancer

1. **Receives Client Requests:** Acts as the first point of contact for incoming requests.
2. **Routes Traffic:** Distributes requests to multiple backend servers.
3. **Monitors Server Health:** Detects failures and reroutes traffic accordingly.
4. **Optimizes Performance:** Uses caching, compression, and other optimizations.

Nginx Load Balancing Mechanisms:

- **Round Robin:** Default method; distributes requests sequentially.
- **Least Connections:** Routes new requests to the least busy server.
- **IP Hash:** Ensures a user is always connected to the same server.

Nginx in Action



This image illustrates how Nginx efficiently manages traffic, ensuring seamless communication between clients and backend servers.

Summary

- **Load Balancing** ensures even distribution of network traffic, improving performance, reliability, and fault tolerance.
- **Nginx** is a powerful web server and reverse proxy that efficiently handles **load balancing, caching, and security**.
- Using **Nginx as a load balancer** optimizes performance, reduces downtime, and enhances scalability for modern web applications.

Nginx remains a top choice for developers and system administrators seeking a **high-performance, reliable, and scalable** web infrastructure solution.

Simple Load balancing Function with LocalStack and Flask

Step-by-Step Guide to AWS VPC, Load Balancer, and Nginx with LocalStack

1. Creating a Virtual Private Cloud (VPC):

To begin, create a VPC using the AWS CLI.

```
aws ec2 create-vpc --cidr-block 10.0.0.0/16 --endpoint-
url=http://localhost:4566
```

2. Creating a Subnet:

Once the VPC is created, define a subnet within it.

```
aws ec2 create-subnet --vpc-id vpc-66375b0cbe498b519
--cidr-block 10.0.1.0/24 --endpoint-url=http://localhost:4566
```

3. Creating an Application Load Balancer:

Create an ALB and associate it with the subnet.

```
aws elbv2 create-load-balancer --name my-load-balancer
--subnets subnet-3f40f7c6e3a26040f --security-groups default
--type application --endpoint-url=http://localhost:4566
```

Setting Up Nginx Containers with Load Balancing

4. Running Backend Containers:

Navigate to the project directory before running containers.

```
cd "C:\Users\rawat\Documents\8 SEMESTER\Cloud
Computing\Lab\Experiment 6\Codes"
```

Run the backend containers using Docker:

```
docker run -d --name backend1 nginx
docker run -d --name backend2 nginx
```

If the image is not found locally, Docker will pull it from the repository.

5. Verify Available Docker Images:

To check the downloaded images:

```
docker images
```

Expected output:

```
REPOSITORY                    TAG       IMAGE ID       CREATED         SIZE
nginx                         latest    b52e0b094bc0   4 weeks ago     192MB
flask-app                     latest    269bf42596ed   3 weeks ago     126MB
...
```

6. Creating and Running a Load Balancer Container:

```
docker run -d --name load-balancer -p 8080:80 -v "C:/Users/rawat/Documents
/8 SEMESTER/Cloud
Computing/Lab/Experiment 6/Codes/nginx.conf:/etc/nginx/nginx.conf" nginx
```

Check logs for potential errors:

```
docker logs load-balancer
```

7. Fixing 'Host Not Found' Error:

The error **"host not found in upstream 'backend1:80'"** means Nginx cannot resolve `backend1`. **All containers must be in the same network.**

Fix: Use a Custom Docker Network

Step 1: Create a Network

```
docker network create my-network
```

Step 2: Start Backend Containers in the Same Network

Remove existing backend containers:

```
docker rm -f backend1 backend2
```

Restart them within the custom network:

```
docker run -d --name backend1 --network my-network nginx
docker run -d --name backend2 --network my-network nginx
```

Step 3: Update `nginx.conf`

Edit your `nginx.conf` file to ensure proper load balancing:

```
events { }

http {
    upstream backend_servers {
        server backend1:80;
        server backend2:80;
    }

    server {
        listen 80;

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

Save the file and restart the load balancer container:

```
docker rm -f load-balancer
docker run -d --name load-balancer --network my-network -p 8080:80 -v
"C:/Users/rawat/Documents/8 SEMESTER/Cloud
Computing/Lab/Experiment 6/Codes/nginx.conf:/etc/nginx/nginx.conf" nginx
```

Step 4: Verify Setup

Check if the load balancer is running properly:

```
docker ps
```

Access the load balancer at:

```
http://localhost:8080
```

If this is not showing up properly, then go to the next step.

Step 5: Restart the Load Balancer

First, remove the existing container:

```
docker rm -f load-balancer
```

Then start it again, **within the same network**:

```
docker run -d --name load-balancer --network my-network -p 8080:80 -v
"C:/Users/rawat/Documents/8 SEMESTER/Cloud
Computing/Lab/Experiment 6/Codes/nginx.conf:/etc/nginx/nginx.conf" nginx
```

Step 6: Verify Everything

Check if containers are running:

```
docker ps
```

Then try opening:

```
http://localhost:8080
```

LocalStack Load Balancer - Flask Simulation

Since Nginx doesn't support load balancing for LocalStack, a Python Flask application is used for round-robin load balancing simulation.

Flask-based Round Robin Load Balancer Simulation

Author: Madhurima Rawat
Date: March 6, 2025
Description:
This script creates a simple web-based simulation of a round-robin load balancer using Flask, Matplotlib, and Seaborn. The application assigns tasks to backend servers in a cyclic manner and visualizes the assigned load using a bar chart.

Usage:

Run the script and access the visualization via:

http://localhost:5000

```
from flask import Flask, render_template_string
import itertools
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np
import io
import base64

app = Flask(__name__)

# Backend servers
backends = ["Backend 1", "Backend 2"]

# Simulated processes
processes = [f"P{i+1}" for i in range(10)]
load_values = np.random.randint(3, 15, size=len(processes))

# Round-robin assignment
backend_cycle = itertools.cycle(backends)
schedule = {process: next(backend_cycle) for process in processes}

@app.route("/")
def home():
    """Renders the home page with the load balancer visualization."""
    try:
        sns.set_theme(style="darkgrid")
        fig, ax = plt.subplots(figsize=(9, 5))

        x_labels = list(schedule.keys())
        y_labels = list(schedule.values())

        custom_colors = [
            "#52796F", "#E63946", "#6A0572", "#457B9D", "#9C6644",
            "#8D99AE", "#2A9D8F", "#E76F51", "#6D597A", "#B56576"
        ]

        bars = ax.bar(x_labels, load_values, color=custom_colors[: len(processes)], width=0.6)

        for bar, (backend, load) in zip(bars, zip(y_labels, load_values)):
            ax.text(
                bar.get_x() + bar.get_width() / 2,
                bar.get_height() + 0.4,
                backend,
                ha="center", fontsize=10, fontweight="bold",
                bbox=dict(facecolor="white", alpha=0.6, edgecolor="black", boxstyle="round,pad
            )
```

```
        ax.set_ylabel("Assigned Load")
        ax.set_title("Round Robin Load Balancer Simulation", fontsize=14, fontweight="bold")
        ax.set_xticklabels(x_labels, rotation=30, ha="right")

        img = io.BytesIO()
        plt.savefig(img, format="png", bbox_inches="tight")
        img.seek(0)
        plt.close(fig)

        image_data = base64.b64encode(img.getvalue()).decode()

        html_template = """
        <!DOCTYPE html>
        <html lang="en">
        <head>
            <meta charset="UTF-8">
            <meta name="viewport" content="width=device-width, initial-scale=1.0">
            <title>Load Balancer</title>
            <style>
                body { font-family: Arial, sans-serif; background-color: #f0f2f5; text-align:
                .container { background: white; padding: 20px; box-shadow: 0 0 15px rgba(0,0,0
                h2 { color: #2c3e50; font-size: 24px; margin-bottom: 10px; }
                p { font-size: 16px; color: #555; margin-bottom: 20px; }
                img { max-width: 100%; border-radius: 10px; margin-top: 10px; }
            </style>
        </head>
        <body>
            <div class="container">
                <h2>Round Robin Load Balancer Simulation</h2>
                <p>
                    Load balancing distributes tasks among multiple backend servers to optimiz
                    This simulation demonstrates a Round Robin approach, where each process is
                </p>
                <img src="{{ image_data }}" alt="Load Balancer Graph">
            </div>
        </body>
        </html>
        """
        return render_template_string(html_template, image_data=image_data)

    except Exception as e:
        return f"Error generating graph: {str(e)}", 500

if __name__ == "__main__":
    app.run(host="0.0.0.0", port=5000)
```

This application displays a bar chart illustrating the distribution of processes across backend servers using the round-robin approach. The graph helps in understanding how the tasks are assigned dynamically.

Run the app using:

```
python app.py
```

Once the server is running, open a web browser and navigate to `http://localhost:5000` to access the application.