Round Robin Server Load Balancer System - Report

Distributed Server Load Balancer System

Team Members

Mohamed Safwat Hassan - 192100140

Abdallah Abdelmonem Abdelaziz - 192100154

Abdelrahman Omar Ali - 191900004

Hamed Mohamed Ragab - 191900056

Ahmed Galal Abdelhay - 192200374

Technical Report

Executive Summary

This report details the architecture and implementation of a distributed server load balancer system built using Flask for the backend and HTML/CSS/JavaScript for the frontend. The system monitors multiple backend servers, distributes incoming traffic among them using a round-robin algorithm, tracks their health status, and visualizes performance metrics through an intuitive dashboard interface.

The project demonstrates key concepts in distributed systems, including:

- Load balancing
- Health monitoring
- Performance metrics collection and visualization
- Fault tolerance
- Real-time data processing

System Architecture

The distributed server load balancer consists of several interconnected components:

1. **Load Balancer** (`load_balancer.py`): The central component that routes incoming requests to available backend servers using a round-robin approach.
2. **Backend Servers** (`server1.py` , `server2.py` , `server3.py` , `server4.py`): Multiple identical servers running on different ports that handle client requests.
3. **Dashboard** (`dashboard.py`): A web application that provides real-time monitoring and visualization of the system status.
4. **Testing Module** (`testing_servers.py`): A script that generates test traffic to demonstrate the load balancer's functionality.
5. **Frontend** (`index.html` , `script.js` , `styles.css`): The user interface for the dashboard, displaying server metrics and visualizations.

Component Analysis

Load Balancer (`load_balancer.py`)

The load balancer serves as the entry point for all client requests and implements several key features:

- **Round-Robin Scheduling**: Uses Python's `itertools.cycle` to distribute incoming requests evenly across available servers.
- **Health Monitoring**: Continuously checks the health of backend servers through a dedicated thread.
- **Request Tracking**: Counts requests sent to each server for load analysis.
- **Metrics Collection**: Gathers and stores performance data for each server, including:

- CPU usage

- RAM usage

- Response latency

- Uptime

- Request count

- **API Endpoints**:

- `/` : Redirects incoming requests to available backend servers

- `/stats` : Provides detailed metrics and status information for all servers

The load balancer also implements Cross-Origin Resource Sharing (CORS) to allow the dashboard to fetch data from its API.

Backend Servers (`serverX.py`)

Each backend server is a Flask application running on a different port (5001-5004) and offers the following endpoints:

- `/` : Returns a simple greeting message
- `/health` : Simple endpoint for health checks, returns "OK" with status code 200
- `/metrics` : Returns detailed server metrics:

- CPU usage (percentage)

- RAM usage (MB)

- Uptime (formatted as HH:MM:SS)

The servers use the `psutil` library to collect system metrics for the specific Python process.

Dashboard (`dashboard.py`)

The dashboard provides a web interface to monitor the status of all backend servers. It includes:

- **API Integration**: Fetches stats from all backend servers and aggregates them
- **Error Handling**: Gracefully handles server failures and timeouts
- **Data Aggregation**: Collects and formats metrics from all servers for the frontend

Testing Module (`testing_servers.py`)

A simple Python script that generates continuous traffic to the load balancer for testing purposes:

- Sends 100 requests with a 2-second delay between requests
- Repeats this process 50 times
- Outputs the status code and response from each request

Frontend Components

HTML Structure (`index.html`):

- Responsive layout with separate sections for different visualizations
- Server status table with dynamic data
- Charts container for visualizing request distribution and server health
- Interactive hover card with detailed server information

JavaScript (`script.js`):

- Fetches server statistics every 2 seconds
- Updates the UI elements in real-time
- Implements three types of visualizations:

1. **Server Status Table**: Shows basic information with color-coded status indicators

2. **Request Distribution Pie Chart**: Visualizes the proportion of requests handled by each server

3. **Health/Requests Line Graphs**: Tracks server health and request count over time

- Interactive elements:

- Hover functionality for detailed server information

- Tooltip positioning that adapts to screen boundaries

Data Flow

1. **Client Request Flow**:

- Client sends request to the load balancer (port 5000)

- Load balancer selects the next available server using round-robin algorithm

- Request is redirected to the selected backend server (ports 5001-5004)

- Backend server processes the request and returns a response

- Response is returned to the client

2. **Monitoring Flow**:

- Dashboard (port 8000) fetches stats from the load balancer's `/stats` endpoint

- Load balancer gathers health and metrics data from each backend server

- Dashboard processes and visualizes this data through the frontend

- Frontend updates every 2 seconds with fresh data

Key Features

1. **Fault Tolerance**:

- The load balancer detects server failures through regular health checks

- Failed servers are excluded from request distribution

- The system continues operating with reduced capacity when servers fail

2. **Dynamic Server Health Monitoring**:

- Continuous background health checks via a dedicated thread

- Real-time updates of server status (UP/DOWN)

- Visualization of server health over time

3. **Performance Metrics**:

- CPU and RAM usage tracking

- Response latency measurement

- Request distribution analysis

- Server uptime monitoring

4. **Interactive Dashboard**:

- Real-time data updates without page refresh

- Multiple visualization types for different metrics

- Detailed server information on hover

- Color-coded status indicators

5. **Historical Data**:

- Time-series graphs showing server health and requests over time

- JSON files stored in `api/stats/` for potential historical analysis

Implementation Details

Round-Robin Load Balancing

The system implements a classic round-robin load balancing algorithm using Python's `itertools.cycle`:

```
server_cycle = itertools.cycle(servers)
```

When a request arrives, the load balancer selects the next server in the rotation with this code:

```
target = next(server_cycle)

if server_status.get(target, False):

    server_request_count[target] += 1
```

```
    return redirect(f"{target}/", code=307)
```

This ensures even distribution of requests among healthy servers.

Health Monitoring

The health check function runs in a separate thread:

```
def health_check():

    while True:

        for server in servers:

            try:

                response = requests.get(server + "/health", timeout=1)

                server_status[server] = (response.status_code == 200)

            except requests.exceptions.RequestException:

                server_status[server] = False

        time.sleep(2)
```

This allows the load balancer to maintain an up-to-date view of server availability without blocking the main request handling thread.

Data Visualization

The frontend implements multiple visualization techniques:

1. **Server Status Table** with visual indicators:

- Green/red dot for UP/DOWN status

- Server URL and request count

2. **Interactive Pie Chart** showing request distribution:

- Color-coded segments for each server

- Consistent colors across refreshes (using URL hash)

- Interactive legends and tooltips

3. **Time-Series Line Graphs** for each server:

- Dual-axis chart showing both requests (left axis) and health status (right axis)

- Color-coded lines (green for requests, red for health)

- Real-time updates with sliding window (last 30 data points)

Testing and Results

The `testing_servers.py` script simulates client traffic by sending 5,000 requests (100 requests × 50 iterations) to the load balancer. With the round-robin algorithm, we expect each server to handle approximately 1,250 requests if all servers remain healthy.

In practice, the distribution may vary slightly due to:

- Server health status changes
- Response time variations
- Network conditions

The dashboard allows visualization of this distribution in real-time, confirming the effectiveness of the load balancing strategy.

Limitations and Future Improvements

1. **Load Balancing Algorithm**:

 - The current implementation uses simple round-robin without considering server load

 - Future improvement: Implement weighted round-robin or least-connections algorithms

2. **Persistence**:

 - The system does not maintain session persistence

 - Future improvement: Add support for sticky sessions with cookie-based routing

3. **Security**:

 - Basic implementation without authentication or HTTPS

 - Future improvement: Add TLS/SSL support and authentication mechanisms

4. **Scalability**:

 - Fixed list of backend servers

 - Future improvement: Implement dynamic server discovery and auto-scaling

5. **High Availability**:

 - Single load balancer instance creates a single point of failure

 - Future improvement: Implement redundant load balancers with failover capability

Conclusion

This distributed server load balancer project successfully demonstrates core principles of distributed systems and load balancing. The combination of Flask-based microservices with a responsive web dashboard provides both functionality and visibility into system operations.

The implementation showcases:

- Effective request distribution across multiple servers
- Real-time monitoring and visualization
- Fault tolerance through health checking
- Performance metrics collection and analysis

While there are several areas for potential improvement, the current system provides a solid foundation for understanding distributed system concepts and load balancing techniques.

Appendix: Project Structure

```
project/

├── api/

│   └── stats/

│       └── [json files for servers status]

├── static/

│   ├── scripts/

│   │   └── script.js

│   └── styles/

│       └── styles.css

├── templates/

│   └── index.html

├── dashboard.py

├── load_balancer.py

├── testing_servers.py

└── serverX.py (4 server instances)
```