

Load Balancer Project - Technical Report

Project Name: Load Balancer **Technology Stack:** Next.js, Node.js, Nginx ,
TypeScript, Docker

By: Abhinav Mathew (B240115CS)

Executive Summary

The Load Balancer project is a comprehensive demonstration of distributed system load balancing with real-time monitoring capabilities. Built using modern web technologies, it features a Next.js frontend dashboard, a Node.js-based round-robin load balancer, automated health monitoring, auto-scaling mechanisms, and nginx reverse proxy configuration. The application demonstrates enterprise-grade load balancing concepts with visual feedback and configurable scaling parameters.

Folder Structure

```
Load-Balancer/
    components.json
    docker-entrypoint.sh
    Dockerfile
    eslint.config.mjs
    next-env.d.ts
    next.config.ts
    nginx.conf
    package.json
    postcss.config.mjs
    README.md
    start.sh
    tsconfig.json
    app/
        globals.css
        layout.tsx
        page.tsx
        scripts/
            config.js
            config.json
            kill-servers.sh
            kill.sh
            launch-servers.sh
            launch.sh
            package.json
            roundRobinAlgorithm.js
            server.js
            singleserver.js
```

```
test-load-balancing.sh
components/
    LetterGlitch.jsx
lib/
    utils.ts
nginx/
    Dockerfile
    nginx.conf
public/
```

Project Architecture Overview

Frontend Architecture (Next.js Dashboard)

The frontend is built using Next.js 16.0.8 with App Router and React 19.2.1, providing a real-time monitoring dashboard with the following characteristics:

- **Framework:** Next.js with App Router architecture
- **Language:** TypeScript for type safety
- **Styling:** Tailwind CSS for responsive design
- **UI Components:** Custom LetterGlitch component for visual effects
- **Real-time Updates:** Live polling of load balancer statistics
- **State Management:** React useState and useEffect hooks

Load Balancer Architecture (Node.js Server)

The core load balancer is a standalone Node.js application with comprehensive features:

- **Algorithm:** Round-robin request distribution
- **Health Monitoring:** Periodic server health checks
- **Auto-scaling:** Intelligent server restart mechanisms
- **Process Management:** Child process spawning for backend servers
- **Statistics Tracking:** Real-time request counting and server metrics

Reverse Proxy Layer (nginx)

nginx acts as the entry point for all client requests:

- **Port Configuration:** Listens on port 80, forwards to load balancer on port 8080
- **Header Management:** Preserves client IP and forwarding information
- **Docker Integration:** Containerized deployment with Node.js services

Core Functionality Analysis

Round-Robin Load Balancing System

Algorithm Implementation (roundRobinAlgorithm.js)

- Maintains a rotating index across healthy servers
- Filters out unhealthy servers from the pool automatically
- Uses http-proxy for seamless request forwarding
- Implements error handling for server failures
- Tracks request statistics for each server

Key Features:

- Automatic unhealthy server exclusion
- Request count tracking per server
- Error handling with graceful fallbacks
- 503 status responses when no servers are available

Health Monitoring and Auto-scaling System

Server Management (server.js)

The load balancer implements sophisticated server lifecycle management:

Health Checks:

- Periodic HTTP requests to verify server responsiveness
- Automatic marking of failed servers as unhealthy
- Configurable health check intervals (default: 2000ms)

Auto-scaling Features:

- Request threshold monitoring (default: 2 requests/interval)
- Automatic server termination when overloaded
- Cooldown periods to prevent rapid restart cycles
- Intelligent server restart with delay mechanisms

Process Control:

- Child process spawning for backend servers
- Process tracking and cleanup
- Graceful server shutdown and restart

Configuration Management

Auto-scaling Configuration (config.js)

```
REQUEST_THRESHOLD: 2 // Requests per interval before scaling action
RESTART_DELAY: 15000 // Cooldown before server restart (15s)
STATS_INTERVAL: 1000 // Statistics update frequency (1s)
HEALTH_CHECK_INTERVAL: 2000 // Health check frequency (2s)
DEBUG_LOGGING: false // Toggle console logging
```

Server Configuration (config.json)

- Defines backend server pool (ports 5001-5005)
- Host configuration for distributed deployments

- Easily extensible for additional servers

Real-Time Dashboard

Frontend Features (page.tsx)

- Live server statistics display
- Request history tracking (last 10 requests)
- Healthy server count monitoring
- Interactive request testing
- Visual effects with LetterGlitch component
- Real-time timestamp updates
- Loading states and error handling

Technical Implementation Details

Development Environment Setup

Concurrent Development: The project uses ‘concurrently’ package to run both frontend dashboard and load balancer simultaneously through the `dev` script, enabling efficient development workflow.

Development Dependencies:

- TypeScript for type safety
- ESLint for code quality
- Tailwind CSS for styling
- Nodemon for backend hot-reload

Containerization Strategy

Docker Configuration:

- Multi-service container with Node.js and nginx
- Alpine Linux base for minimal footprint
- Exposed ports: 80 (nginx), 3000 (Next.js), 8080 (Load Balancer)
- Automated startup script orchestration

Production Considerations

The current configuration is optimized for demonstration with intentionally low thresholds. For production deployment:

1. **Increase Request Threshold** from 2 to 1000-10000+ requests
2. **Reduce Restart Delay** from 15 seconds to 2-5 seconds
3. **Implement Exponential Backoff** for repeated failures
4. **Add Session Persistence** for stateful applications
5. **Enable HTTPS/TLS** termination
6. **Scale Backend Server Pool** based on capacity requirements

Key Features Summary

Implemented Features

- **Round-robin load balancing** with automatic server rotation
- **Health monitoring** with automatic failure detection
- **Auto-scaling** with configurable thresholds and cooldowns
- **Real-time dashboard** with live statistics and monitoring
- **nginx reverse proxy** for production-like architecture
- **Docker containerization** for easy deployment
- **Configurable parameters** for different environments
- **Request history tracking** for debugging and monitoring
- **Visual feedback** with custom UI components
- **Error handling** with graceful degradation

Usage Workflow

System Startup:

1. Execute `bash start.sh` to launch all services
2. nginx starts on port 80 as reverse proxy
3. Load balancer initializes on port 8080
4. Backend servers spawn on ports 5001-5005
5. Next.js dashboard launches on port 3000

Load Balancing Operation:

1. Client requests hit nginx on port 80
2. nginx forwards to load balancer on port 8080
3. Load balancer applies round-robin algorithm
4. Requests distributed across healthy backend servers
5. Health checks monitor server responsiveness
6. Auto-scaling triggers server restarts when overloaded
7. Dashboard provides real-time monitoring and statistics

Monitoring and Management:

1. Access dashboard at `http://localhost:3000`
2. View real-time server statistics and health status
3. Test load balancing with interactive request buttons
4. Monitor request history and server performance
5. Observe auto-scaling behavior during load testing

Conclusion

The Load Balancer project successfully demonstrates enterprise-grade load balancing concepts with modern web technologies. The implementation showcases proper distributed system architecture, real-time monitoring capabilities, automated scaling mechanisms, and production-ready deployment strategies.

The project serves as an excellent foundation for understanding load balancing algorithms, health monitoring systems, auto-scaling mechanisms, and full-stack development with Next.js and Node.js. The modular architecture, comprehensive configuration options, and Docker containerization provide a solid foundation for production deployment with appropriate scaling considerations and performance optimizations.

The combination of nginx reverse proxy, Node.js load balancer, and React dashboard creates a complete load balancing solution that can be adapted for real-world production environments with minimal modifications to the core architecture.