IT5551

COMPUTER NETWORKS

A MINI PROJECT REPORT

Dynamic Load Balancing System for Distributed Servers

Submitted by

Gowtham Rajasekaran 2022506084

B. Tech (5/8)

DEPARTMENT OF INFORMATION TECHNOLOGY

MADRAS INSTITUTE OF TECHNOLOGY

ANNA UNIVERSITY- CHENNAI

CHENNAI - 600 044

August / December 2024

Dynamic Load Balancing System for Distributed Servers

AIM:

The aim of this project is to create a load balancer that efficiently distributes client requests across multiple servers based on the server's current load. The load balancing is achieved by calculating the "load factor," which is used to select the server with the least load for each incoming request. This system ensures optimal utilization of server resources and prevents overloading any single server.

OVERVIEW:

This mini-project involves creating a client-server application with a load balancer in Java. The project includes four main components:

1. **Server.java**: This program handles the server functionalities. It maintains a synchronized method to ensure that only one thread accesses the function at a time.
2. **Client.java**: The client program simulates multiple client threads that send requests to the server.
3. **LoadBalancer.java**: Implements a load balancing algorithm that assigns requests to servers based on their load factor. The load factor is the current load divided by the server's total capacity.
4. **LoadBalancerGUI.java**: The graphical user interface (GUI) is built using Swings and AWT. It visualizes the server loads and allows the user to control the simulation.

SCOPE:

- **Server Load Management**: The project focuses on efficient load balancing by distributing client requests across multiple servers based on their current load.
- **Real-time Simulation**: During runtime, the system simulates client requests being sent to servers and dynamically assigns requests to servers with lower load factors.

- **Multithreading**: The project involves multiple threads, where client threads repeatedly send requests, and server threads handle these requests.
- **Dynamic Server Capacity**: Server capacities are generated randomly at runtime, adding variability to the simulation.

DESCRIPTION:

This project implements a **Dynamic Load Balancing System** that efficiently distributes client requests across multiple servers in a network. The system ensures optimal server utilization by assigning incoming client requests to the server with the least load, based on a "load factor" metric. The load factor is calculated as the ratio of the current load to the total capacity of a server.

The project consists of four main components:

1. **Server Program (Server.java)**: The server handles client requests and ensures thread safety by using synchronized methods to prevent concurrent access issues.
2. **Client Program (Client.java)**: Multiple client threads are simulated to generate requests and send them to the load balancer for distribution to the servers.
3. **Load Balancer (LoadBalancer.java)**: This component contains the core load balancing algorithm. The load balancer continuously monitors server loads and routes incoming requests to the server with the lowest load factor.
4. **Graphical User Interface (LoadBalancerGUI.java)**: Built using Sphinx and AWT, the GUI provides a visual representation of the load distribution across the servers. It allows the user to start and stop the simulation while displaying the real-time status of each server.

The project demonstrates the application of multithreading, resource management, and load balancing techniques, making it a practical solution for ensuring fair and efficient resource allocation in a distributed network environment.

CODE:

Server.java

- Contains methods that handle requests from clients.
- Synchronized methods ensure thread safety (i.e., only one thread accesses the function at a time).

```
import javax.swing.*;

public class Server implements Runnable {
    private final int serverId;
    private final int weight;
    private int currentLoad = 0;
    private JTextArea logArea;
    private JProgressBar progressBar;
    private volatile boolean running = true; // Flag to control thread execution

    public Server(int serverId, int weight, JTextArea logArea, JProgressBar progressBar)
{
        this.serverId = serverId;
        this.weight = weight;
        this.logArea = logArea;
        this.progressBar = progressBar;
    }

    public synchronized int getCurrentLoad() {
        return currentLoad;
    }

    public synchronized void handleRequest(int requestWeight) {
        currentLoad += requestWeight;
        updateProgressBar();

        logArea.append("Server " + serverId + " handling request. Current Load: " +
currentLoad + "\n");
        new Thread(() -> {
            try {
                Thread.sleep(requestWeight * 1000);  // Simulate processing time
                finishRequest(requestWeight);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }).start();
    }
    private synchronized void finishRequest(int requestWeight) {
        currentLoad -= requestWeight;
        updateProgressBar();
        logArea.append("Server " + serverId + " finished request. Current Load: " +
currentLoad + "\n");
    }
```

```
    private void updateProgressBar() {
        int progress = (int) ((currentLoad / (double) weight) * 100);
        progressBar.setValue(progress);
        progressBar.setString("Load: " + currentLoad + "/" + weight);
    }

    public void stop() {
        running = false; // Set the flag to false
    }

    @Override
    public void run() {
        while (running) {
            try {
                Thread.sleep(1000);  // Keep the server alive
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        logArea.append("Server " + serverId + " has stopped.\n");
    }

    public int getWeight() {
        return weight;
    }
}
```

Client.java

- Simulates multiple client threads generating requests.
- Each client thread sends requests to the load balancer, which then assigns it to the appropriate server.

```
import javax.swing.*;
import java.util.Random;

public class Client implements Runnable {
    private LoadBalancer loadBalancer;
    private JTextArea logArea;
    private volatile boolean running = true; // Flag to control thread execution

    public Client(LoadBalancer lb, JTextArea logArea) {
        this.loadBalancer = lb;
        this.logArea = logArea;
    }

    @Override
    public void run() {
        try {
            Random rand = new Random();
            while (running) {
```

```
                int requestWeight = rand.nextInt(10) + 1;
                logArea.append("Client sending request with weight: " + requestWeight +
"\n");
                loadBalancer.sendRequest(requestWeight);
                Thread.sleep(rand.nextInt(3000) + 1000);
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        logArea.append("Client has stopped sending requests.\n");
    }

    public void stop() {
        running = false; // Set the flag to false
    }
}
```

LoadBalancer.java

- The core load balancing algorithm is implemented here.
- The algorithm checks each server's load factor (current load / total capacity) and assigns the client request to the server with the lowest load factor.

```
import java.util.List;

public class LoadBalancer {
    private final List<Server> servers;

    public LoadBalancer(List<Server> servers) {
        this.servers = servers;
    }
    public synchronized void sendRequest(int requestWeight) {
        // Find the server with the least weighted connections
        Server bestServer = null;
        double minLoadFactor = Double.MAX_VALUE;

        for (Server server : servers) {
            double loadFactor = (double) server.getCurrentLoad() / server.getWeight();
            if (loadFactor < minLoadFactor) {
                minLoadFactor = loadFactor;
                bestServer = server;
            }
        }
        if (bestServer != null) {
            bestServer.handleRequest(requestWeight);
        } else {
            System.out.println("No available server to handle the request.");
        }
    }
}
```

LoadBalancerGUI.java

- A graphical interface that visualizes the servers' status (e.g., capacity and current load).
- Provides buttons to start and stop the simulation.
- The GUI shows real-time data of server loads and client requests.

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.util.Random;
import java.util.ArrayList;
import java.util.List;

public class LoadBalancerGUI {
    private JFrame frame;
    private JTextArea logArea;
    private List<Server> servers;
    private LoadBalancer loadBalancer;
    private List<JProgressBar> serverProgressBars;
    private List<Client> clients; // Store client instances
    private Random random;

    public LoadBalancerGUI() {
        random = new Random();
        initialize();
        setupLoadBalancer();
    }

    private void initialize() {
        frame = new JFrame("Load Balancer Simulation");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(800, 600);
        frame.setLayout(new BorderLayout());

        logArea = new JTextArea();
        logArea.setEditable(false);
        logArea.setLineWrap(true);
        logArea.setWrapStyleWord(true);
        frame.add(new JScrollPane(logArea), BorderLayout.CENTER);

        // Panel for server load indicators
        JPanel serverPanel = new JPanel();
        serverPanel.setLayout(new GridLayout(3, 2));  // 3 servers, 2 columns
        serverProgressBars = new ArrayList<>();

        for (int i = 0; i < 3; i++) {
            JPanel panel = new JPanel();
            panel.setLayout(new BorderLayout());
            JLabel label = new JLabel("Server " + (i + 1));
```

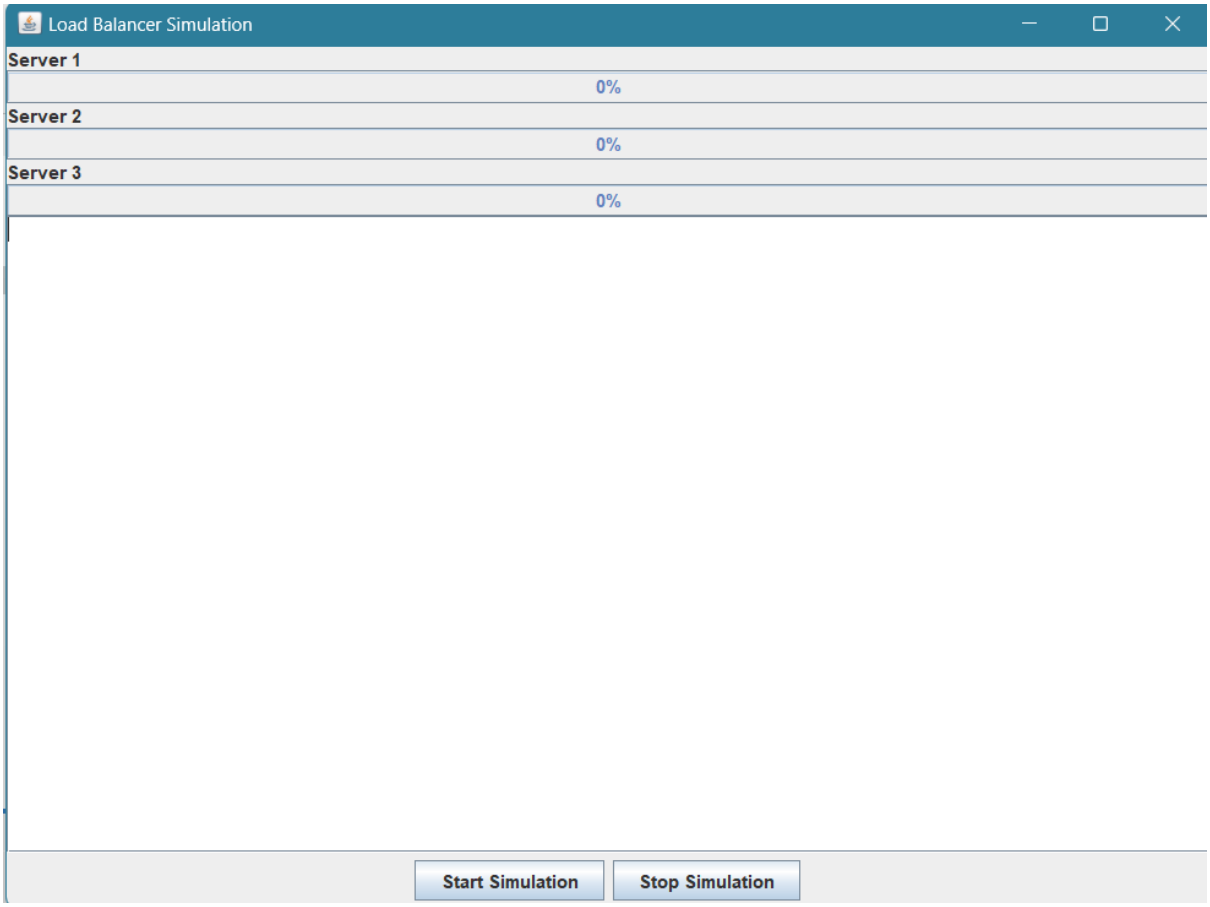
```
            JProgressBar progressBar = new JProgressBar();
            progressBar.setStringPainted(true);
            serverProgressBars.add(progressBar);
            panel.add(label, BorderLayout.NORTH);
            panel.add(progressBar, BorderLayout.CENTER);
            serverPanel.add(panel);
        }

        frame.add(serverPanel, BorderLayout.NORTH);

        // Button panel
        JPanel buttonPanel = new JPanel();
        JButton startButton = new JButton("Start Simulation");
        JButton stopButton = new JButton("Stop Simulation");

        buttonPanel.add(startButton);
        buttonPanel.add(stopButton);
        frame.add(buttonPanel, BorderLayout.SOUTH);

        // Action listeners for buttons
        startButton.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                startSimulation();
            }
        });

        stopButton.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                stopSimulation();
            }
        });

        frame.setVisible(true);
    }

    private void setupLoadBalancer() {
        servers = new ArrayList<>();
        servers.add(new Server(1, random.nextInt(10) + 1, logArea,
serverProgressBars.get(0)));  // Server 1
        servers.add(new Server(2, random.nextInt(10) + 1, logArea,
serverProgressBars.get(1))); // Server 2
        servers.add(new Server(3, random.nextInt(10) + 1, logArea,
serverProgressBars.get(2)));  // Server 3

        loadBalancer = new LoadBalancer(servers);
        clients = new ArrayList<>(); // Initialize the clients list
    }

    private void startSimulation() {
        for (Server server : servers) {
```

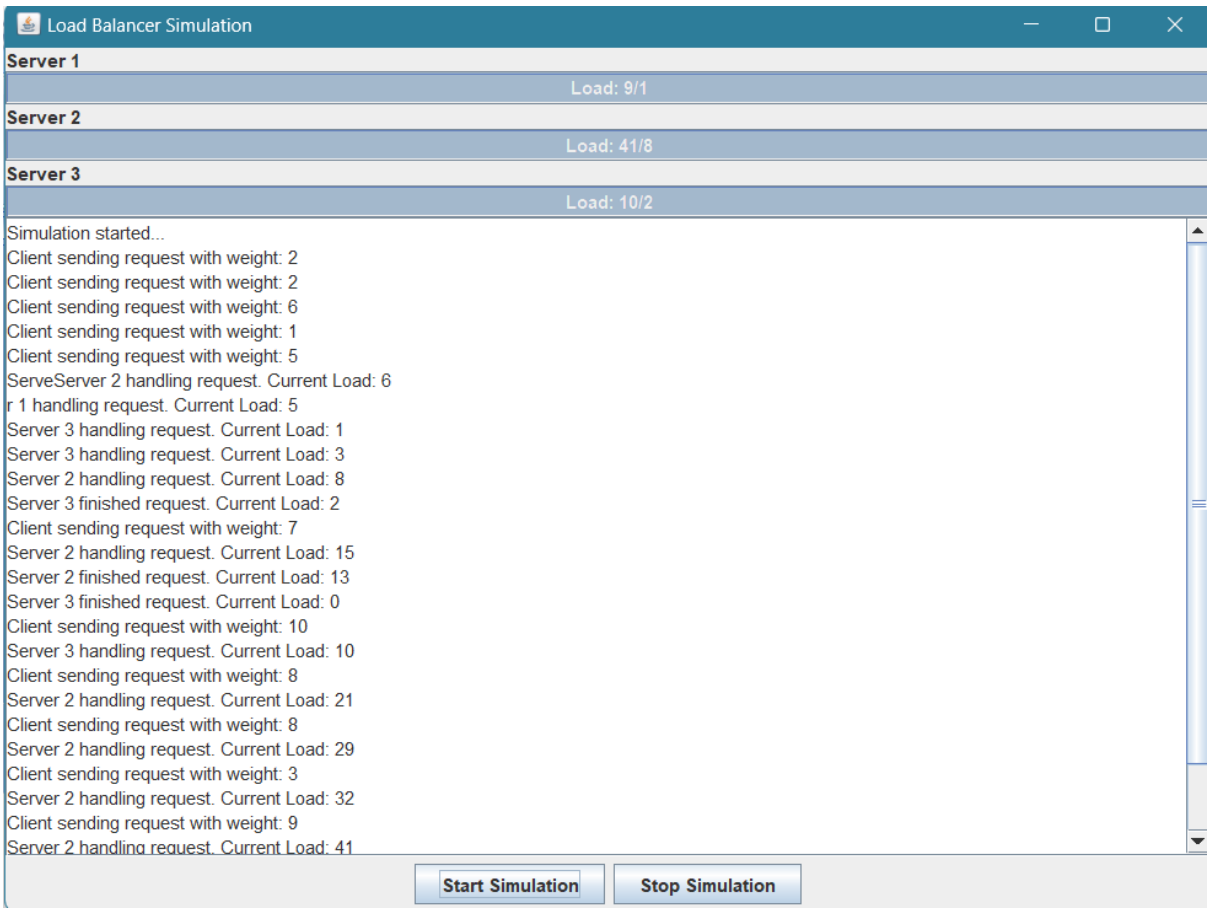
```
            new Thread(server).start();
        }
        // Start clients in separate threads
        for (int i = 0; i < 5; i++) {  // Example with 5 clients
            Client client = new Client(loadBalancer, logArea);
            clients.add(client); // Store the client instance
            new Thread(client).start();
        }
        logArea.append("Simulation started...\n");
    }

    private void stopSimulation() {
        logArea.append("Stopping simulation...\n");
        // Stop all clients
        for (Client client : clients) {
            client.stop();
        }
        // Stop all servers
        for (Server server : servers) {
            server.stop();
        }
        logArea.append("Simulation stopped.\n");
    }

    public static void main(String[] args) {
        SwingUtilities.invokeLater(LoadBalancerGUI::new);
    }
}
```

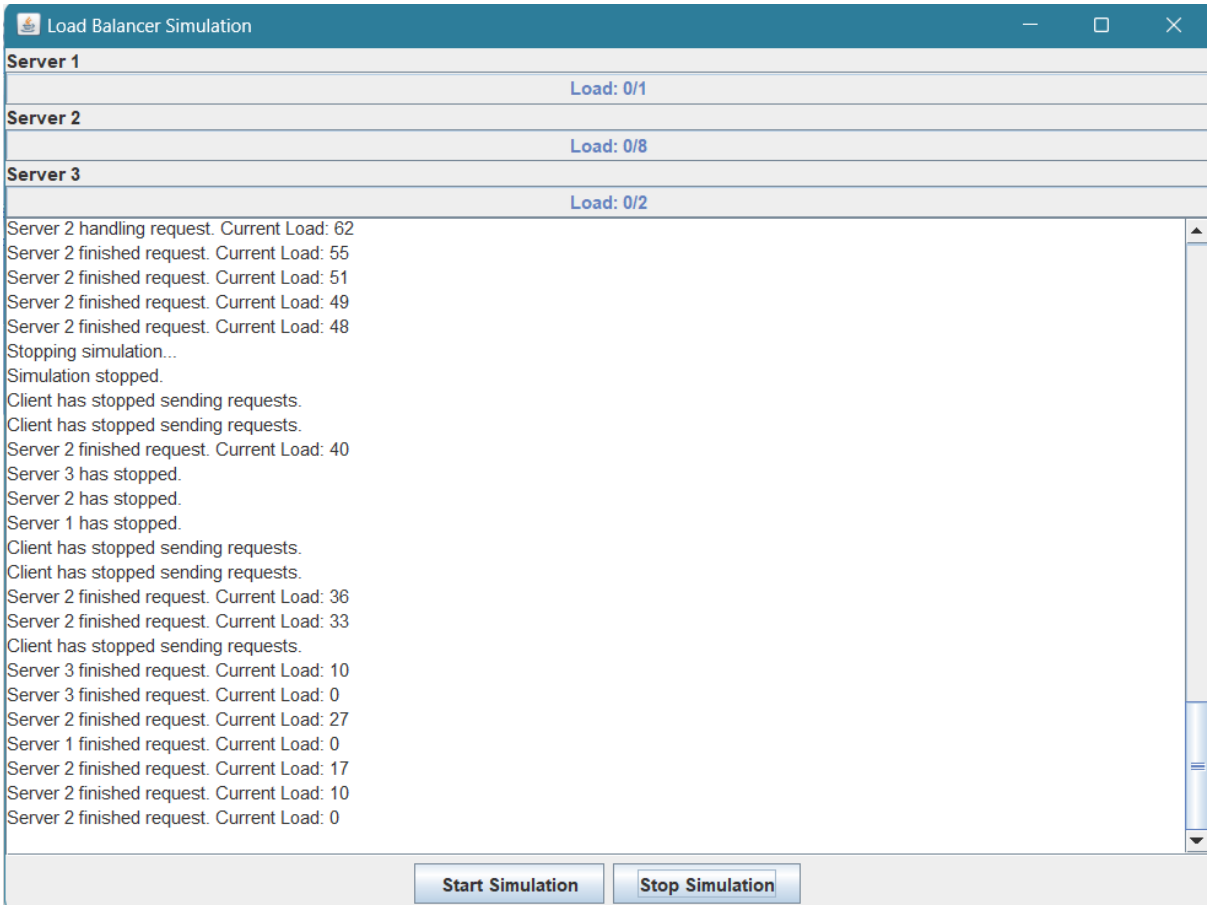
OUTPUT:

When the user interacts with the **LoadBalancerGUI.java**, the following will happen:

1. **Start Simulation**: Client threads start generating requests. Servers begin to handle requests based on the load balancing algorithm.
2. **Stop Simulation**: The client stops generating requests, and the server clears any existing requests. Servers go idle.
3. **Graphical Representation**: The GUI displays three bars representing the servers, where each bar's height is based on the server's load (current load versus total capacity). This helps to visualize the load balancing process in real-time.


Load Balancer Simulation
Server 1
0%
Server 2
0%
Server 3
0%
Start Simulation
Stop Simulation


Load Balancer Simulation
Server 1
Load: 9/1
Server 2
Load: 41/8
Server 3
Load: 10/2
Simulation started...
Client sending request with weight: 2
Client sending request with weight: 2
Client sending request with weight: 6
Client sending request with weight: 1
Client sending request with weight: 5
ServeServer 2 handling request. Current Load: 6
r 1 handling request. Current Load: 5
Server 3 handling request. Current Load: 1
Server 3 handling request. Current Load: 3
Server 2 handling request. Current Load: 8
Server 3 finished request. Current Load: 2
Client sending request with weight: 7
Server 2 handling request. Current Load: 15
Server 2 finished request. Current Load: 13
Server 3 finished request. Current Load: 0
Client sending request with weight: 10
Server 3 handling request. Current Load: 10
Client sending request with weight: 8
Server 2 handling request. Current Load: 21
Client sending request with weight: 8
Server 2 handling request. Current Load: 29
Client sending request with weight: 3
Server 2 handling request. Current Load: 32
Client sending request with weight: 9
Server 2 handling request. Current Load: 41
Start Simulation
Stop Simulation



CONCLUSION:

This mini-project demonstrates how load balancing works in a network environment by distributing client requests based on server load. By calculating and comparing the load factor of each server, the system dynamically assigns requests to the least-loaded server, improving the overall efficiency and preventing overloads. The graphical user interface offers a user-friendly way to visualize the operation of the system, making it easier to understand the load distribution process in real-time. This project serves as a practical implementation of the core concepts of load balancing, multithreading, and network resource management.