VIT Vidyalankar Institute of Technology Accredited A+ by NAAC	DEPARTMENT OF COMPUTER ENGINEERING

Experiment No. 02

Semester	B.E. Semester VIII – Computer Engineering
Subject	Distributed Computing Lab
Subject Professor In-charge	Dr. Umesh Kulkarni
Lab Professor In-charge	Prof. Prakash Parmar
Academic Year	2024-25

Student Name	Chaitanya Mandale
Roll Number	21102A0044

Title: To Implement a Distributed application using socket -- Application consists of a server which takes an integer value from the client, calculates factorial and returns the result to the Client program.

Explanation:

Distributed computing involves dividing a task or system into multiple components, which are executed across different machines to achieve scalability, reliability, and efficiency. In such architectures, several key roles and components work together to handle computational workloads effectively.

1. Role of the Client

The client is the **entry point** into the distributed system and is responsible for initiating requests.

- **Responsibilities of the Client:**
    - **Request Generation:** Sends tasks (e.g., data or computation requests) to the system for processing.

    - **Transparency:** Does not need to know the specifics of how tasks are executed or which server is performing them.
    - **Receive Responses:** Retrieves results after processing is completed.
    - **Stateless Interaction:** Often, the client is designed to remain stateless, meaning it does not store information about previous interactions with the system, enabling better scalability.

- **Examples of Clients:**
    - A web browser sending HTTP requests to a web server.

2. Role of Load Balancer (Middleware Layer)

The load balancer is a critical intermediary in distributed systems that ensures optimal distribution of incoming requests across multiple servers or workers.

- **Responsibilities of the Load Balancer:**
    - **Task Distribution:** Distributes client requests among available servers or workers to prevent overloading any single node.
    - **Dynamic Resource Allocation:** Monitors available resources (e.g., servers, workers) and adjusts routing based on system status.
    - **Fault Tolerance:** Detects and bypasses non-functional or overloaded nodes to maintain system reliability.
    - **Scalability:** Ensures the system can handle more requests by adding more servers or workers dynamically.

- **Load Balancing Strategies:**
    - **Round-Robin:** Requests are distributed sequentially among servers in a cyclic order.
    - **Random Selection:** A random server is chosen for each request.

3. Role of Workers (or Servers)

Workers (or servers) are the **execution nodes** that perform the actual computation or service requested by the client.

- **Responsibilities of Workers:**
    - **Task Execution:** Perform the computational or processing tasks (e.g., calculating factorials, serving web pages, querying databases).
    - **Scalability:** New workers can be added to handle higher workloads dynamically.
    - **Fault Tolerance:** If a worker fails, the load balancer redirects tasks to other available workers.
    - **Autonomy:** Each worker operates independently, allowing for modularity and resilience.

4. Shared Registry or Coordination Service

In distributed systems, a **shared registry** or **coordination service** is often used to manage information about available workers or resources.

- **Purpose:**
    - Acts as a central point for resource discovery.
    - Maintains metadata about active servers, their health, and availability.
- **Examples of Coordination Mechanisms:**
    - **File-Based:** A simple text file listing available workers (as in this experiment).

5. Key Features of This Architecture

- **Decoupling:** The client does not directly interact with workers but goes through the load balancer. This ensures that the client is unaffected by changes in worker configurations.
- **Fault Tolerance:** If a worker becomes unavailable, the load balancer redirects tasks to

other workers, ensuring continuous operation.

- **Scalability:** New workers can be added dynamically to handle increased workloads without disrupting the existing system.
- **Transparency:** The client and workers are unaware of the load balancing strategy, simplifying their roles.

Implementation :

client.py :

```
import socket
def start_client():
    host = "127.0.0.1"
    port = 65432
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as client_socket:
        client_socket.connect((host, port))
        number = input("Enter an integer to calculate its factorial: ")
        client_socket.sendall(number.encode())
        result = client_socket.recv(1024)
        print(f"Factorial received: {result.decode()}")
if __name__ == "__main__":
    start_client()
```

load_balancer.py

```
import socket
import threading
REGISTRATION_FILE = "worker_ports.txt"
worker_index = 0
```

```
def get_available_workers():
    with open(REGISTRATION_FILE, "r") as file:
        lines = file.readlines()
    workers = [("127.0.0.1", int(port.strip())) for port in lines]
    return workers
def handle_client(client_socket):
    global worker_index
    data = client_socket.recv(1024).decode()
    print(f"Received request for factorial of {data}")
    workers = get_available_workers()
    if not workers:
        print("No available workers.")
        client_socket.sendall(b"Error: No workers available.")
        client_socket.close()
        return
    worker = workers[worker_index]
    worker_index = (worker_index + 1) % len(workers)
    try:
        with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as worker_socket:
            worker_socket.connect(worker)
            worker_socket.sendall(data.encode())
            result = worker_socket.recv(1024)
        client_socket.sendall(result)
        print(f"Response sent to client: {result.decode()}")
    except Exception as e:
        print(f"Error communicating with worker {worker}: {e}")
        client_socket.sendall(b"Error: Worker communication failed.")
    finally:
        client_socket.close()
def start_load_balancer():
```

```
    host = "127.0.0.1"
    port = 65432
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as server_socket:
        server_socket.bind((host, port))
        server_socket.listen()
        print(f"Load Balancer listening on {host}:{port}")
        while True:
            client_socket, addr = server_socket.accept()
            print(f"Client connected: {addr}")
            threading.Thread(target=handle_client,
                             args=(client_socket,)).start()
if __name__ == "__main__":
    start_load_balancer()
```

worker_server.py

```
import socket
import os
from math import factorial
REGISTRATION_FILE = "worker_ports.txt"
def register_worker(port):
    with open(REGISTRATION_FILE, "a") as file:
        file.write(f"{port}\n")
def start_worker():
    host = "127.0.0.1"
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as server_socket:
        server_socket.bind((host, 0))
        port = server_socket.getsockname()[1]
        register_worker(port)
        print(f"Worker registered on port {port}")
        server_socket.listen()
```

```
        print(f"Worker listening on {host}:{port}")
        while True:
            conn, addr = server_socket.accept()
            with conn:
                print(f"Connected by {addr}")
                data = conn.recv(1024)
                if not data:
                    break
                number = int(data.decode())
                print(f"Worker received number: {number}")
                result = factorial(number)
                conn.sendall(str(result).encode())
                print(f"Worker sent factorial: {result}")
if __name__ == "__main__":
    if not os.path.exists(REGISTRATION_FILE):
        open(REGISTRATION_FILE, "w").close()
    start_worker()
```

Explanation for Implementation :

Here we are starting server workers (actual computational instances) on random available ports and registering that port numbers in a file. This file is accessed by load balancer to get available worker information via port numbers. Load balancer then routes client request in round-robin manner to available workers.

Output :

Client Sending 3 Requests to Application (Load Balancer) :

```
PS C:\Users\chait\OneDrive\Documents\Lone\DCPracs\Implementation\Lab2> python .\client.
py
Enter an integer to calculate its factorial: 12
Factorial received: 479001600
PS C:\Users\chait\OneDrive\Documents\Lone\DCPracs\Implementation\Lab2> python .\client.
py
Enter an integer to calculate its factorial: 15
Factorial received: 1307674368000
PS C:\Users\chait\OneDrive\Documents\Lone\DCPracs\Implementation\Lab2> python .\client.
py
Enter an integer to calculate its factorial: 7
Factorial received: 5040
```

Load Balancer Handling 3 Received Requests :

```
PS C:\Users\chait\OneDrive\Documents\Lone\DCPracs\Implementation\Lab2> python .\load_ba
lancer.py
Load Balancer listening on 127.0.0.1:65432
Client connected: ('127.0.0.1', 50744)
Received request for factorial of 12
Response sent to client: 479001600
Client connected: ('127.0.0.1', 50887)
Received request for factorial of 15
Response sent to client: 1307674368000
Client connected: ('127.0.0.1', 50890)
Received request for factorial of 7
Response sent to client: 5040
```

Server Workers Actually Computing Results :

```
PS C:\Users\chait\OneDrive\Documents\Lone\DCPracs\Implementation\Lab2> python .\server_
worker.py
Worker registered on port 50737
Worker listening on 127.0.0.1:50737
Connected by ('127.0.0.1', 50886)
Worker received number: 12
Worker sent factorial: 479001600
```

```
PS C:\Users\chait\OneDrive\Documents\Lone\DCPracs\Implementation\Lab2> python .\server_
worker.py
Worker registered on port 50739
Worker listening on 127.0.0.1:50739
Connected by ('127.0.0.1', 50888)
Worker received number: 15
Worker sent factorial: 1307674368000
```

```
PS C:\Users\chait\OneDrive\Documents\Lone\DCPracs\Implementation\Lab2> python .\server_
worker.py
Worker registered on port 50741
Worker listening on 127.0.0.1:50741
Connected by ('127.0.0.1', 50892)
Worker received number: 7
Worker sent factorial: 5040
```

Conclusion :

This experiment demonstrates the effectiveness of a distributed computing architecture with a load balancer and multiple workers. The load balancer ensures efficient request distribution using a round-robin strategy, improving scalability and fault tolerance. Workers dynamically register themselves, allowing flexibility and reducing manual configuration. This system highlights the benefits of modularity, scalability, and resilience, making it well-suited for real-world applications requiring efficient resource utilization and high availability.