Assignment No. 08

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

Student Name	Deep Salunkhe
Roll Number	21102A0014

Title: Distributed load-balancer system

System Overview

This distributed load-balancer system is designed to efficiently distribute incoming computational tasks across multiple servers in a network while maintaining high availability, scalability, and fault tolerance. The system will handle dynamic server loads, server failures, and varying task requirements.

Core Components

1. Load Balancer Nodes

- **Primary Load Balancer**: The initial entry point for all incoming tasks
- **Secondary Load Balancers**: Hot standby instances for failover
- **Regional Load Balancers**: For geographically distributed deployments

2. Worker Nodes

- **Compute Servers**: Execute the actual tasks
- **Resource Monitors**: Track CPU, memory, network, and other metrics
- **Heartbeat Agents**: Regularly report node health to load balancers

3. Control Plane

- **Configuration Manager**: Stores and distributes system configuration
- **Service Registry**: Maintains the list of available worker nodes
- **Metrics Collector**: Aggregates performance data from all nodes
- **Scheduler**: Makes task assignment decisions based on algorithms

4. Data Storage

- **Task Queue**: Holds pending tasks awaiting assignment
- **Result Store**: Stores completed task outputs
- **Logging System**: Records system events for monitoring and debugging

Architecture Diagram

```
[Client Requests] → [Primary Load Balancer]
                        |
                        v
                [Task Queue] ↔ [Control Plane]
                        |
                        v
              [Worker Node 1]   [Worker Node 2]   [Worker Node N]
                  (Healthy)         (Overloaded)      (Failed)
```

Load Balancing Algorithms

1. Round Robin

- Simple rotation through available servers
- Pros: Easy to implement, works well with homogeneous tasks
- Cons: Doesn't account for server load or capacity

2. Least Connections

- Directs new tasks to the server with fewest active connections
- Pros: Adapts to varying task durations

- Cons: Doesn't consider server capacity differences

3. Weighted Algorithms

- Assigns weights based on server capacity (CPU, memory, etc.)
- Can combine with Round Robin or Least Connections
- Pros: Accounts for heterogeneous infrastructure
- Cons: More complex configuration

4. Resource-Based Scheduling

- Considers CPU load, memory usage, network latency
- Uses real-time metrics to make decisions
- Pros: Most efficient resource utilization
- Cons: Higher overhead from monitoring

5. Consistent Hashing

- Useful for stateful applications where tasks must stick to servers
- Minimizes redistribution when nodes join/leave
- Pros: Good for caching scenarios
- Cons: Not optimal for purely computational loads

Fault Tolerance Mechanisms

1. Heartbeat Monitoring

- Worker nodes send regular heartbeats to control plane
- Missing heartbeats trigger failover procedures

2. Task Retry Logic

- Failed tasks are re-queued with attempt counters
- Exponential backoff for retry intervals

3. Redundant Storage

- Task queue and results stored with replication

- Consensus protocols (Raft/Paxos) for critical metadata

4. Graceful Degradation

- System continues operating with reduced capacity during failures
- Critical tasks prioritized when resources are constrained

Task Lifecycle

1. **Submission**: Client submits task with optional priority/requirements
2. **Queueing**: Task enters prioritized queue in load balancer
3. **Scheduling**: Load balancer selects optimal worker based on algorithm
4. **Execution**: Worker processes task and reports progress
5. **Completion**: Results stored and client notified
6. **Cleanup**: Resources released, logs archived

Performance Considerations

Monitoring Metrics

- Task completion rate
- Average task duration
- Queue wait times
- Resource utilization across nodes
- Error/failure rates

Scaling Strategies

- **Vertical Scaling**: Increase capacity of existing nodes
- **Horizontal Scaling**: Add more worker nodes dynamically
- **Autoscaling**: Automatically adjust worker pool based on load

Implementation Options

Communication Protocols

- REST/HTTP for management interfaces

- gRPC for high-performance internal communication
- WebSockets for real-time progress updates

Technology Choices

- **Containerization**: Docker/Kubernetes for worker isolation
- **Service Mesh**: Istio/Linkerd for advanced traffic management
- **Message Queues**: RabbitMQ/Kafka for task distribution

Advanced Features (Optional)

1. **Predictive Scaling**: Use historical data to anticipate load spikes
2. **Task Prioritization**: QoS tiers for different task types
3. **Energy-Aware Scheduling**: Optimize for power efficiency
4. **Multi-Cloud Support**: Distribute across cloud providers
5. **A/B Testing**: Route subsets of tasks differently for experimentation

This design provides a comprehensive framework for implementing a distributed load-balancer system that can handle computational workloads efficiently while maintaining reliability and scalability. The modular architecture allows for different algorithms and components to be swapped based on specific requirements.