PROJECT PROPOSAL

Distributed Task Queue System

A Scalable Job Processing Architecture using Node.js

Subject:	Parallel & Distributed Computing
Technology:	Node.js + Redis
Duration:	~20 Hours
Type:	Academic Project

1. Introduction

A Distributed Task Queue System is a software architecture pattern that enables multiple worker processes to execute tasks from a shared queue. This pattern is fundamental in modern distributed systems and is used by companies like Netflix, Uber, and Slack to handle millions of background jobs daily.

This project demonstrates core concepts of parallel and distributed computing including load balancing, fault tolerance, and worker coordination in a practical, real-world application.

2. Problem Statement

Traditional single-threaded applications struggle with time-consuming tasks like image processing, email sending, or data analysis. These tasks block the main application, causing poor user experience and inefficient resource utilization. A distributed task queue solves this by:

- Offloading heavy tasks to background workers
- Distributing work across multiple processes/machines
- Providing fault tolerance and retry mechanisms
- Enabling horizontal scaling of workloads

3. Objectives

- Build a functional task queue with producer-consumer architecture
- Implement multiple worker processes for parallel task execution
- Demonstrate load balancing across workers
- Add fault tolerance with task retry mechanisms
- Create a monitoring dashboard to visualize system performance

4. System Architecture

4.1 Components

Component	Description	Technology
Task Producer	Creates and submits tasks to queue	Express.js API
Message Queue	Stores pending tasks	Redis / In-Memory
Worker Pool	Multiple processes executing tasks	Node.js Cluster
Result Store	Stores completed task results	Redis / JSON
Dashboard	Monitors queue and workers	Socket.io + HTML

4.2 Data Flow

1. Client sends task request to Producer API $\rightarrow$ 2. Producer adds task to Redis queue $\rightarrow$ 3. Available Worker picks up task (FIFO) $\rightarrow$ 4. Worker processes task and updates status $\rightarrow$ 5. Result stored and client notified

5. Technologies Used

Technology	Purpose
Node.js	Runtime environment for JavaScript
Redis	In-memory queue and pub/sub messaging
Express.js	REST API for task submission
Cluster Module	Spawning multiple worker processes
Socket.io	Real-time dashboard updates
UUID	Unique task identification

6. Key Distributed Computing Concepts

6.1 Producer-Consumer Pattern

Producers create tasks without waiting for completion. Consumers (workers) independently pull and process tasks. This decoupling allows both to scale independently.

6.2 Load Balancing

Tasks are distributed evenly across available workers. When a worker is busy, tasks wait in queue until a worker becomes available, ensuring optimal resource utilization.

6.3 Fault Tolerance

If a worker crashes during task execution, the task is automatically re-queued. Failed tasks are retried with exponential backoff to handle transient failures.

6.4 Horizontal Scaling

Adding more workers increases throughput linearly. The system can scale from 2 workers to hundreds without architectural changes.

7. Implementation Plan

Phase	Tasks	Hours
Phase 1: Setup	Project structure, dependencies, basic queue	3
Phase 2: Core	Worker pool, task processing, Redis integration	6
Phase 3: Features	Retry logic, priority queues, task status	5
Phase 4: Dashboard	Real-time monitoring UI with Socket.io	4
Phase 5: Testing	Load testing, documentation, demo	2
	Total	**20**

8. Expected Outcomes

- Functional distributed task queue handling 100+ concurrent tasks
- Demonstrated speedup with multiple workers vs single worker
- Real-time dashboard showing queue depth and worker status
- Successful fault recovery when workers are terminated
- Performance metrics comparing different worker configurations

9. Real-World Applications

This architecture is used in production systems worldwide:

- Email Services: Sending bulk emails without blocking
- Image Processing: Resizing, thumbnails, filters
- Video Transcoding: Converting video formats
- Report Generation: Creating PDFs, Excel exports
- Data Pipelines: ETL jobs, data synchronization

10. Conclusion

This project provides hands-on experience with distributed computing concepts that are essential for modern software development. By building a task queue from scratch, we gain deep understanding of message passing, worker coordination, and fault-tolerant system design. The skills learned are directly applicable to real-world systems used by tech companies globally.

References

- Node.js Cluster Documentation - nodejs.org/api/cluster.html
- Redis Documentation - redis.io/documentation
- Bull Queue Library - github.com/OptimalBits/bull
- Distributed Systems Patterns - martinfowler.com