Distributed Task Queue System

BISHAL K C, Tennessee Technological University, USA
PRADIP KUNWAR, Tennessee Technological University, USA

1 Problem Statement

Modern computational workloads such as data analytics, scientific computation, and large-scale simulations often require executing thousands of independent or long-running tasks efficiently. Traditional centralized systems struggle to handle such workloads due to limited processing capacity, single points of failure, and difficulty in scaling dynamically with demand.

Our project aims to design and implement a fault-tolerant distributed task processing system that addresses these challenges by distributing computational workloads across multiple worker nodes. The system will dynamically assign tasks to available workers, continuously monitor worker health through heartbeat mechanisms, and automatically recover from node failures by reassigning interrupted tasks to healthy workers. The architecture will support horizontal scaling by allowing new worker nodes to join the system seamlessly, enabling the system to adapt to varying workload demands without requiring downtime or manual reconfiguration.

Why Distributed System?

We adopt a distributed architecture for the following key reasons:

(1) **Scalability:** Distributed systems can scale horizontally by adding more worker nodes. As workload increases, the system accommodates additional computation without performance loss. In our project, this scalability ensures that as tasks grow, more workers can be seamlessly integrated, enabling efficient handling of dynamic workloads.
(2) **Availability & Fault Tolerance:** Unlike centralized architectures that fail when a single node crashes, distributed systems ensure continuity through redundancy and recovery. In our implementation, the Dispatcher monitors worker health via heartbeat signals and reassigns tasks from failed nodes, preventing data loss and minimizing downtime.
(3) **Asynchronicity:** Distributed task processing uses asynchronous communication, allowing workers to operate independently without blocking others. This non-blocking design improves throughput and resource utilization. In our project, it enables efficient parallel task handling, resulting in faster completion times and smoother coordination across all nodes.

2 System Architecture

This project proposes a custom-built, fault-tolerant distributed task processing system composed of two main components: a central **Dispatcher** and a dynamic pool of **Worker Nodes** as shown in figure 1. The design follows a pull-based model, emphasizing scalability, fault tolerance, and consistency. [1–4]

Authors' Contact Information: Bishal K C, Tennessee Technological University, Cookeville, Tennessee, USA; Pradip Kunwar, Tennessee Technological University, Cookeville, Tennessee, USA.

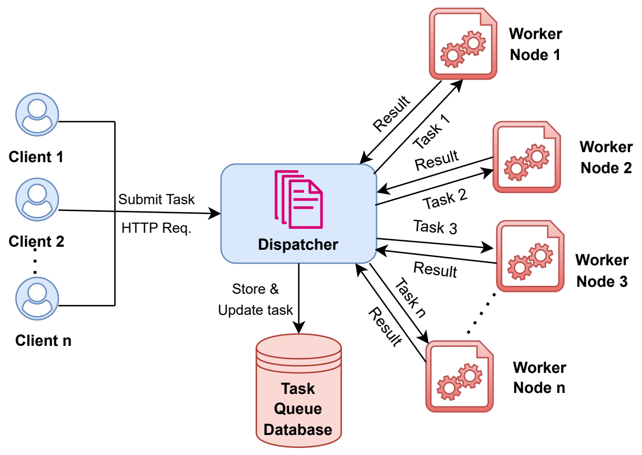

Fig. 1. Distributed Task Queue System Architecture

2.1 Components & Data Flow:

- **Dispatcher (Coordinator):** The central control node responsible for managing the entire task lifecycle. It exposes REST APIs for task submission, assignment, heartbeats, and result collection.
- **Task Queue Database:** A persistent database co-located with the Dispatcher, storing all task records and their states (*pending, in-progress, completed*). It serves as the single source of truth for the system.
- **Worker Nodes:** Independent processes that periodically poll the Dispatcher for available tasks, execute assigned computations, and report results asynchronously.

2.2 Data Flow Overview

The following steps explains how tasks move through the system, from user submission to completion and result collection:

(1) A user submits a new task to the Dispatcher via the API.
(2) The Dispatcher logs it in the Task Queue Database as *pending*.
(3) An idle Worker polls the Dispatcher for work and receives the task.
(4) The Dispatcher updates the task to *in-progress* and records a heartbeat timestamp.
(5) The Worker executes the computation, periodically sending heartbeat signals.
(6) Upon completion, the Worker submits the result back to the Dispatcher, which marks the task as *completed.*

2.3 Scalability, Fault Tolerance, and Consistency:

This section outlines how the proposed architecture ensures efficient scaling, resilient fault recovery, and consistent task management across all distributed components:

(1) **Scalability:** New Workers can be added at any time, enabling horizontal scaling to meet higher workloads.

(2) **Fault Tolerance:** The Dispatcher monitors worker health via heartbeats. If a worker fails, its task is reassigned to another node. In case of Dispatcher downtime, Workers cache completed results locally and resubmit when connectivity is restored.
(3) **Consistency:** The Task Database enforces a consistent view of all task states, ensuring at-least-once execution. Tasks are designed to be idempotent to handle rare reassignments safely.

3 Technology Stack

In this project, we plan to use a lightweight, reliable, and easily deployable technology stack to build a fault-tolerant distributed task processing system.

(1) **Programming Language:** Python will be used for its simplicity, concurrency support, and rich ecosystem for distributed systems development.
(2) **Frameworks and Libraries:** Flask will provide the REST API for Dispatcher endpoints, while `asyncio` and `requests` (or `httpx`) will enable efficient asynchronous communication and task handling between nodes.
(3) **Database:** SQLite or PostgreSQL will serve as the central Task Database to maintain consistency and reliability, with SQLite optionally used for lightweight local testing.
(4) **Communication Protocol:** RESTful HTTP will be used to ensure simple, language-agnostic communication between the Dispatcher and Workers, supporting easy debugging and scalability.
(5) **Deployment:** The system will be containerized with Docker and can be deployed on AWS or Google Cloud, allowing seamless scaling by adding new Worker containers.
(6) **Version Control:** Git and GitHub will be used for source management and collaborative development.

4 Development Timeline

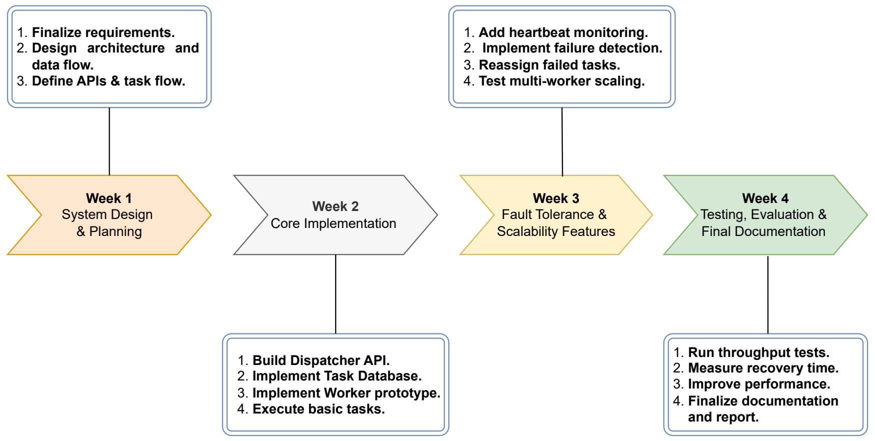
1. Finalize requirements.
2. Design architecture and data flow.
3. Define APIs & task flow.
1. Add heartbeat monitoring.
2. Implement failure detection.
3. Reassign failed tasks.
4. Test multi-worker scaling.
Week 1 System Design & Planning
Week 2 Core Implementation
Week 3 Fault Tolerance & Scalability Features
Week 4 Testing, Evaluation & Final Documentation
1. Build Dispatcher API.
2. Implement Task Database.
3. Implement Worker prototype.
4. Execute basic tasks.
1. Run throughput tests.
2. Measure recovery time.
3. Improve performance.
4. Finalize documentation and report.

Fig. 2. Four-Week Development Timeline

Our project is planned to be completed over four weeks, starting November 10 and ending December 8. Each week is a distinct phase with clear goals, as shown in Figure 2.

5 Evaluation Criteria

The success of the proposed distributed task processing system will be evaluated based on its ability to reliably execute tasks, efficiently scale with increased workloads, and maintain robust fault-tolerance under node failures or network disruptions. These criteria collectively measure how well the system meets its design goals of performance, resilience, and consistency in a distributed environment.

Performance Metrics:

- **Throughput:** The system's efficiency will be measured by the number of tasks completed per minute.
- **Scalability:** Throughput will be evaluated as the number of Worker nodes increases (e.g., 1, 2, 4) to verify near-linear scaling and the system's ability to handle larger workloads.
- **Fault Recovery:** The total recovery time will be measured when one or more Worker nodes fail. This includes reassignment of pending tasks and restoration of full system operation, with recovery time recorded relative to the number of failed nodes.

Testing Strategy:

- Perform multiple concurrent task submissions to assess throughput and task completion latency.
- Simulate controlled Worker failures to evaluate recovery time, fault-tolerance, and task reassignment effectiveness.
- Incrementally add Worker nodes to test horizontal scalability and system performance under increased load.

6 Risk Assessment

The proposed distributed task processing system faces several potential risks. Each risk, its impact, and the mitigation strategy are outlined below:

- **Worker Failure:** A Worker may crash mid-task, interrupting execution. The Dispatcher monitors heartbeats and will reassign any incomplete tasks to healthy Workers to ensure continuity.
- **Dispatcher as a Single Point of Failure (SPOF):** If the Dispatcher goes down, new tasks cannot be assigned. Workers store completed results locally and synchronize once the Dispatcher returns. In production, a high-availability cluster would be needed.
- **Duplicate Task Execution:** A slow Worker may be marked dead and its task reassigned, causing two Workers to complete the same task. This is mitigated by designing tasks to be idempotent, so repeated execution does not affect correctness.
- **Database Overload:** Frequent updates can overload the Task Database, reducing performance. Batch updates and connection pooling will help manage the load efficiently.

7 Summary

In summary, our goal is to build a simple yet reliable fault-tolerant distributed task-processing system that can execute workloads across multiple worker nodes while ensuring reliability, fast recovery, and seamless horizontal scaling.

References

[1] GeeksforGeeks. 2025. Distributed Task Queue - Distributed Systems. https://www.geeksforgeeks.org/system-design/distributed-task-queue-distributed-systems/. Last updated: July 23, 2025; Accessed: November 10, 2025.

[2] Theodore Johnson. 1995. Designing a distributed queue. In *Proceedings. Seventh IEEE Symposium on Parallel and Distributed Processing*. IEEE, 304–311.

[3] Veljko Maksimović, Miloš Simić, Milan Stojkov, and Miroslav Zarić. 2023. Task queue implementation for edge computing platform. In *Conference on Information Technology and its Applications*. Springer, 471–480.

[4] System Design Handbook. [n. d.]. Design a Distributed Job Scheduler: System Design Guide. https://www.systemdesignhandbook.com/guides/design-a-distributed-job-scheduler/. Accessed: November 10, 2025.