# Design Document for Distributed Task Scheduler

## Overview

The distributed task scheduler is designed to allow clients to register one-time and recurring tasks that are executed at specified times. The system ensures that tasks are picked up and executed within 10 seconds of their scheduled time. The design prioritizes simplicity, high availability, durability, and scalability.

## High-Level Architecture

The system comprises the following main components:

1. Task Scheduler Service
2. Task Executor Service
3. Task Storage
4. Client Interface (API Gateway)
5. Monitoring and Logging Service

## Component Breakdown

### Task Scheduler Service

#### **Responsibilities**:

* Receive and store tasks (one-time and recurring) from clients.
* Parse cron expressions for recurring tasks.
* Determine the next execution time for tasks.
* Publish tasks to a message queue close to their execution time.

#### **Technology Choices:**

* Language: Node.js and Express.
* Framework: Node-cron (for Node.js).

#### **Design Decisions:**

* Using a distributed message queue ensures scalability and fault tolerance.

### Task Executor Service

#### **Responsibilities**:

* Subscribe to the message queue to receive tasks ready for execution.
* Execute the tasks within the specified execution window (10 seconds).
* Log the execution details.

#### **Technology Choices:**

* Language: Node.js and Express.

#### **Design Decisions:**

* Decoupling task execution from scheduling allows independent scaling of these components based on load.
* A logging service ensures that execution details are recorded for durability and traceability.

### Task Storage

#### **Responsibilities**:

* Store task definitions and schedules.
* Ensure data durability and high availability.

#### **Technology Choices:**

* Database: PostgreSQL (for relational data).

#### **Design Decisions:**

* A durable and highly available database ensures tasks are not lost.
* ACID properties handle concurrent task updates and reads efficiently.

### Client Interface (API Gateway)

#### **Responsibilities**:

* Provide a RESTful API for clients to register, edit, delete, and view tasks.

#### **Technology Choices:**

* Language: Node.js and Express with React.

#### **Design Decisions:**

* A RESTful API simplifies client integration and ensures stateless interactions.

### Monitoring and Logging Service

#### **Responsibilities**:

* Monitor system health and task execution metrics.
* Log task executions and errors.

#### **Technology Choices:**

* Monitoring: Node.js and Express.
* Logging: Basic logging within the console for simplicity.

#### **Design Decisions:**

* Future enhancements can include integration with monitoring tools like Prometheus and Grafana for metrics, and ELK stack for centralized logging.

## Communication Flow

### Task Registration:

* Clients interact with the API Gateway to register tasks.
* The API Gateway validates and forwards task details to the Task Scheduler Service.
* The Task Scheduler Service stores the task in the Task Storage and schedules it.

### Task Scheduling and Execution:

* The Task Scheduler Service determines the next execution time and publishes the task to the message queue.
* The Task Executor Service subscribes to the message queue, picks up tasks for execution, and logs the execution details.

### Monitoring and Logging:

* All components emit logs and metrics to the Monitoring and Logging Service.
* System administrators can view real-time metrics and logs through dashboards.

## Scaling and Chokepoints

### **Scaling:**

* **Horizontal Scaling:** Both the Task Scheduler and Task Executor Services can be scaled horizontally by adding more instances.
* **Database:** Implement read replicas and partitioning to handle increased load.

### **Chokepoints**:

* **Database:** As the number of tasks grows, the database may become a bottleneck. This can be mitigated by sharding and using read replicas.
* **Task Executor:** With many tasks, executors may become overwhelmed. Horizontal scaling and task prioritization can help manage load.

### **Cost-Effectiveness**

* Using open-source technologies like RabbitMQ, Celery, PostgreSQL, and Prometheus ensures cost-effectiveness.
* Deploying services on container orchestration platforms like Kubernetes allows efficient resource utilization and scaling.

## **Conclusion**

This design document outlines a straightforward yet robust distributed task scheduler system. The critical design decisions and tradeoffs focus on achieving high availability, durability, and scalability while ensuring the system remains cost-effective. The design is modular, allowing each component to scale independently and handle failures gracefully, ensuring reliable task scheduling and execution.