

**Distributed Task Assigning Management System**

**COSC-6352- ADVANCED OPERATING SYSTEMS**

**Professor - Yongzhi Wang**

**Table of Contents:  
1. Introduction**

**3. System Function Description**

**4. System Architecture**

**5. Distributed System Features**

**6. Reflections**

**1.Introduction:**

This document outlines the development and functionality of a real-time task management application designed to facilitate the scheduling and monitoring of tasks using a distributed system. The primary objective of this system is to provide a robust platform where tasks can be dynamically added, processed, and tracked through various statuses, leveraging Node.js, Redis, and Docker technologies. By implementing a modern backend architecture and a straightforward user interface, the application aims to enhance productivity and operational efficiency in task handling. This report details the components, functionalities, and technologies employed, along with insights into the system architecture and performance enhancements.

**3. System Function Description**

This application is crafted to streamline task management processes, enabling users to add, monitor, and retrieve tasks with efficiency and precision. Below, we delve deeper into each core functionality:

**Task Addition**:

Users initiate task creation by submitting a form via the application’s web interface. This action sends a POST request to the /task endpoint. Each task is captured with a unique identifier and a specific description, which is then serialized into JSON format and stored in Redis with an initial status of 'Added'. This setup allows for rapid addition and retrieval, leveraging Redis’s high-performance data handling capabilities.

A screenshot of a task operations dashboard

Description automatically generated

A screenshot of a computer

Description automatically generated

*Challenges:*

* Concurrency Management: Ensuring that multiple users can submit tasks simultaneously without overloading the system or causing data loss.
* Data Validation: Verifying that task IDs are unique and descriptions are correctly formatted to prevent errors.

*Solutions:*

* Asynchronous Handling: Using Node.js's non-blocking I/O to handle multiple submissions efficiently.
* Redis Transactions: Utilizing Redis's atomic capabilities to ensure data is written correctly even under load.
* Input Validation: Implementing checks within the server code to validate input before processing.

**Task Status Monitoring**:

As tasks enter the processing queue, their status progresses through predefined stages. Initially marked as 'Added', tasks are automatically updated to 'Processed' after a predetermined interval, simulating task handling. This progression is managed asynchronously, using background worker processes that monitor task completion and update their status in the Redis store. This real-time updating is crucial for maintaining an accurate and current view of task progress, essential for operational management and user feedback.

A screenshot of a computer

Description automatically generated

*Challenges:*

* Status Synchronization: Keeping the task status updated across different stages of processing without delays or inconsistencies.
* Scalability: Ensuring the system can handle a large number of tasks being updated regularly.

*Solutions:*

* Queue Management: Using Bull, a robust queueing system, to manage task status updates in a controlled and efficient manner.
* Clustering: Leveraging Node.js clustering to distribute task processing across multiple CPU cores, enhancing the system’s ability to scale.

**Task Querying:**

The application provides a robust querying capability via the /tasks endpoint, where users can request lists of tasks based on their current status. This feature is implemented using HTTP GET requests that retrieve task data from Redis, filtered according to the specified status. It supports real-time operational oversight and enhances user interaction by providing immediate visibility into the state of various tasks.

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

*Challenges:*

* Query Performance: Maintaining fast response times as the volume of tasks grows.
* Data Filtering: Efficiently filtering tasks based on status without excessive computation.

*Solutions:*

* Indexed Searches: Configuring Redis to handle indexed searches efficiently, allowing for quicker retrieval based on task status.
* Optimized Filtering: Implementing server-side optimizations to process and return filtered task data swiftly.

**Logging System Activity**:

Monitoring system operations is facilitated through the /system-logs endpoint. This function aggregates and displays logs related to task processing and system interactions. Each task’s lifecycle events, from creation to completion, are logged in real-time, offering valuable insights into system performance and aiding in troubleshooting. This logging capability is integral to maintaining system reliability and accountability.

These enhanced functionalities ensure that the system not only meets the needs of real-time task management but also provides a user-friendly and responsive environment for managing a wide array of tasks efficiently.

A screenshot of a computer

Description automatically generated

*Challenges:*

* Log Volume Management: Handling large volumes of log data without impacting system performance.
* Real-time Access: Providing real-time logging information without delays.

*Solutions:*

* Separate Logging Service: Using a dedicated service or module to handle logging independently from task processing, minimizing impact on core functionalities.
* Incremental Updates: Implementing techniques such as log rotation and incremental log updates to manage log size and accessibility.

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

**4. System Architecture**

The architecture of the task management system is meticulously designed to support robust, scalable, and real-time operations. Here we explore each component in detail, highlighting their functionalities and interactions:

**A diagram of a software process

Description automatically generated**

**1.Frontend Design**:

The frontend of the application is crafted using HTML and JavaScript. It utilizes asynchronous JavaScript (AJAX) to communicate with the backend, allowing for dynamic content updates without page reloads. This approach ensures a seamless user experience where task updates and statuses are reflected instantly on the user interface.

**Code:**

A screenshot of a computer program

Description automatically generated

**2.Backend Architecture: Node.js and Express**

Node.js, the backend leverages the Express framework to handle API requests efficiently. Express facilitates quick setup of routing and middleware, making it easier to manage the flow of data and handle requests for task addition, updating, and retrieval. The backend serves as the nerve center of the application, orchestrating the interaction between the frontend, Redis database, and task processing queues.

**Code:**

A screen shot of a computer program

Description automatically generated

**3.Redis Data Storage**:

Redis is employed as the main data store, renowned for its speed and efficiency in handling key-value data. It stores all task-related information, including unique identifiers, descriptions, and current statuses. The choice of Redis is strategic for achieving minimal response times in data retrieval and update operations, which is crucial for real-time systems.

**Code:**

**A screenshot of a computer program

Description automatically generated**

**4.Task Queue Management with Bull**:

Bull is integrated on top of Redis to manage task queues. It regulates the lifecycle of each task from initiation to completion, ensuring tasks are processed orderly and reliably. This queuing system is essential for controlling the workflow within the system, managing task priorities, and ensuring consistency and reliability in task execution.

**Code:  
  
Queue setup (queue.js):**

**A screenshot of a computer

Description automatically generated**

**Adding to queue (producer.js):**

**A screen shot of a computer

Description automatically generated**

**5.Worker Process Management**:

Dedicated worker processes are configured to handle the actual task processing. These workers are responsible for fetching tasks from the Bull queue, processing them according to business logic (such as updating task statuses or performing computations), and then marking them as complete. This decoupling of task processing from the main application logic allows for more efficient resource management and scalability.

A screenshot of a computer program

Description automatically generated

**6.Clustering for Enhanced Performance:**

The application utilizes Node.js’s cluster module to distribute the load across multiple CPU cores. This method improves the application’s capacity to handle simultaneous requests and enhances overall system resilience and efficiency.

A screen shot of a computer

Description automatically generated

**7.Docker-based Deployment and Scalability**:

The entire application is containerized using Docker, which encapsulates each component in its own container, streamlining deployment and scalability. This containerized setup supports a microservices architecture by allowing individual components like the web server, Redis, and worker processes to be scaled independently based on demand.

**Microservices Structure**: Adopting a microservices architecture, the system ensures that different functionalities (like task processing and data storage) are isolated yet operate cohesively. This separation facilitates easier maintenance, better fault isolation, and enhanced scalability.

This detailed architecture not only supports high availability and scalability but also ensures the system is robust enough to manage high volumes of tasks efficiently while maintaining real-time performance.

Docker-compose.yml

A screenshot of a computer program

Description automatically generated

Docker file

A screenshot of a computer program

Description automatically generated

**5. Distributed System Features**

The architecture of the task management system incorporates several key features to ensure it operates effectively in a distributed environment. These features address scalability, fault tolerance, and the capacity for real-time updates:

**5.1. Scalability:**

**Overview:** One of the primary features of this system is its ability to scale horizontally. Using Docker containers, the application can deploy multiple instances of the server and Redis database across different machines or cloud environments. This scalability is critical during peak loads, allowing the system to add resources dynamically and maintain performance without degradation. Additionally, the Node.js cluster module facilitates load distribution across multiple CPU cores, enhancing the ability to handle more concurrent users and tasks.

A screenshot of a computer

Description automatically generated

**5.2.Fault Tolerance**:

**Overview:**

The system is designed to be fault-tolerant, with several mechanisms in place to handle failures gracefully. Redis, used for data storage, supports persistence and can recover its state after a crash. The Bull queue, built on Redis, ensures that messages (tasks) are not lost even if a processing worker fails; it can reassign the task to another worker for processing. Moreover, the application's architecture allows failed components, like a single Docker container, to be restarted automatically without affecting the overall system availability.

**A screenshot of a computer

Description automatically generated**

the failed task is automatically assigned to task queue

**A screenshot of a computer program

Description automatically generated**

**5.3. Real-Time Updates**:

**Overview:**

To manage real-time updates efficiently, the system utilizes a well-designed message queuing system (Bull) that ensures all task updates are promptly communicated to the front end. This feature is crucial for maintaining a consistent and synchronous view of tasks across all client interfaces. The real-time capability is further supported by using lightweight operations in Redis, which can quickly update and broadcast task status changes with minimal latency.

**A screenshot of a computer

Description automatically generated**

**5.4.Distributed Caching**:

**Overview:**  
Redis also acts as a distributed cache, storing frequently accessed data like task statuses and details, which reduces the load on the main database and improves response times for user queries. This caching mechanism is vital for enhancing read performance and reducing latency, especially important in systems with a high volume of read operations.

**A screenshot of a computer

Description automatically generated**

**5.5 Data Integrity and Consistency:**

Ensuring data integrity and consistency in a distributed environment is managed through transactional support in Redis. Transactions ensure that all operations involved in a particular task update are completed atomically, preventing data corruption and ensuring consistency across the system.

**6.Challenges and Solutions**

In developing this task management system, several challenges were encountered, particularly regarding the distributed nature of the application, task processing reliability, and real-time data synchronization. Here’s how each challenge was addressed:

**1.Challenge: Ensuring Consistent Access Across Multiple Server Ports**

* 1. **Solution:** To maintain uniform access to the task queue across various server ports, the system was set up to connect every server instance to a common Redis instance. This was achieved by configuring the connection details through environmental variables. Such a configuration ensures that tasks added from any server port are accessible and managed consistently.

**2.Challenge: Reliable Task Assignment and Processing**

* 1. **Solution:** The system utilizes the Bull queue library built on Redis to manage task assignments and processing reliably. This library ensures that tasks are processed sequentially and includes built-in mechanisms to retry tasks that initially fail, thus preventing tasks from being lost due to processing errors.

**3.Challenge: Worker Node Efficiency and Scalability**

* 1. **Solution:** Worker nodes are configured to handle tasks independently from the primary server application. This separation allows for the scaling of processing capabilities as needed without affecting the performance of the main application. The Node.js clustering module is employed to distribute the workload effectively across multiple CPU cores, significantly enhancing the system's capacity to manage large volumes of tasks.

**4.Challenge: Handling and Reassigning Failed Tasks**

* 1. **Solution:** The system incorporates a comprehensive error-handling strategy within the worker processes. If a task fails, it automatically triggers a retry mechanism. Continuous failures are recorded and queued for detailed analysis. This approach helps ensure that temporary disruptions do not lead to permanent losses of tasks, facilitating ongoing system maintenance and problem resolution.

**5.Challenge: Accurate and Real-time System Logging**

* 1. **Solution:** All interactions and status updates related to tasks are meticulously logged via the /system-logs endpoint, which retrieves the data directly from Redis. This method ensures that every alteration to task status is recorded in real time, providing a precise and up-to-date view of the system’s operations. This is essential for effective troubleshooting and conducting system audits.