TaskForge

Abstract

This project introduces a robust distributed task scheduler designed to overcome the limitations of traditional cron-based systems, which are typically bound to single machines. By leveraging a distributed architecture, the scheduler enhances fault tolerance and scalability, ensuring high availability and reliability across multiple nodes. The system utilises Docker for containerisation, gRPC for efficient inter-service communication, and PostgreSQL for persistent storage. This approach not only mitigates the risks associated with single points of failure but also supports complex scheduling requirements in large-scale, distributed environments.

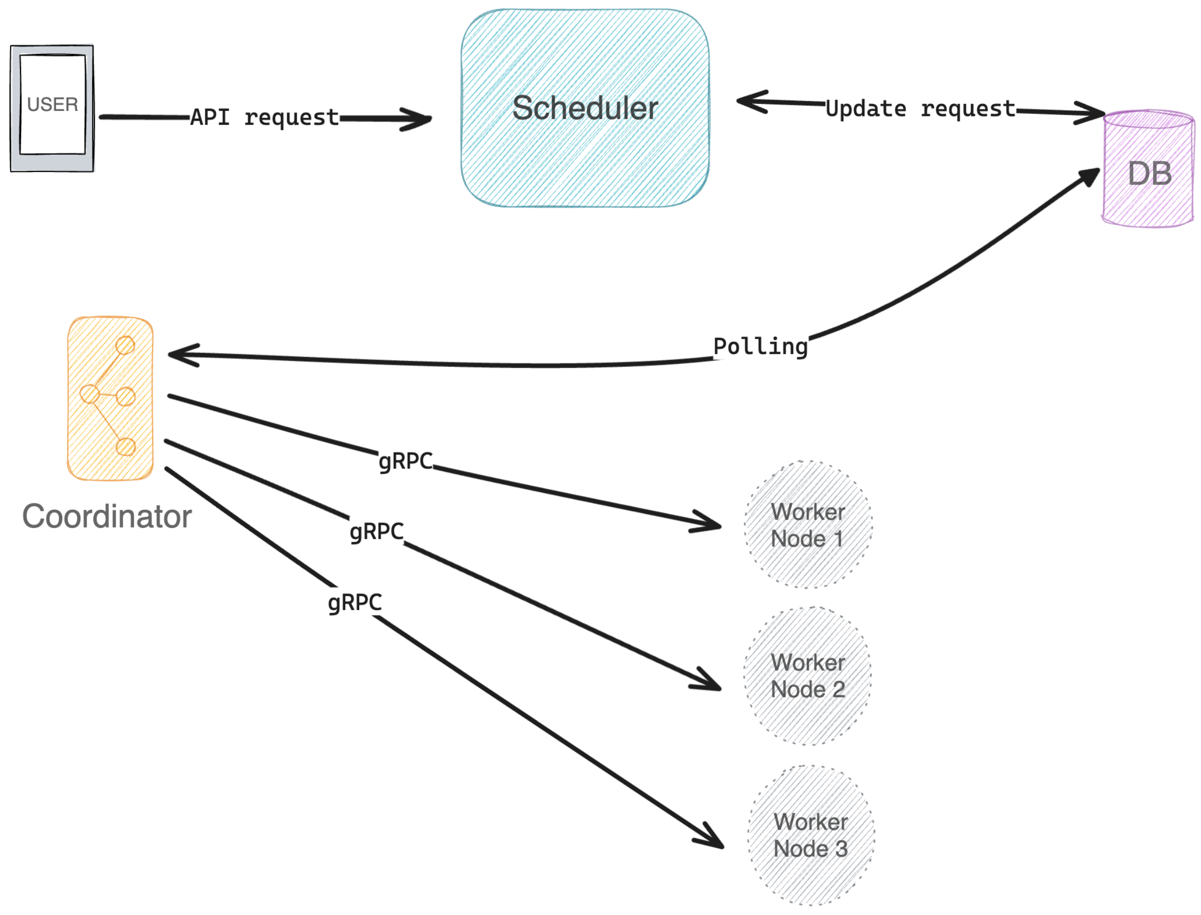
Introduction

Task scheduling is an essential function in various computing environments, automating the execution of scripts and commands at predetermined times. Traditional scheduling solutions, such as Unix cron, are inherently limited by their dependency on single servers, which introduces risks of downtime and scalability challenges. In response to these limitations, this project develops a distributed task scheduler that decentralises the scheduling mechanism across multiple nodes. This design is particularly advantageous for environments requiring high availability and the ability to scale dynamically in response to workload variations. The system is built using contemporary technologies: Docker for deploying and managing isolated environments, gRPC for low-latency communication between services, and PostgreSQL for managing state with high reliability. The scheduler is designed to be resilient, with mechanisms to handle failures gracefully and ensure that tasks are consistently executed on time, even in the event of node failures or network issues.

System Architecture

The system is composed of several key components:

- **PostgreSQL Database**: Stores tasks and their execution states.
- **Scheduler**: Interfaces with users for task submission.
- **Coordinator**: Manages task distribution and worker node coordination.
- **Worker Nodes**: Execute the tasks assigned by the coordinator.



The distributed task scheduler system is designed to handle task scheduling across multiple nodes using a combination of Docker containers, each serving a specific role within the architecture. Here's a concise overview of how these components interact:

User Interaction and Task Submission

Users interact with the system by sending API requests to the **Scheduler** service. These requests include the command to be executed (`cmd`) and the time the task is scheduled to execute. The Scheduler, upon receiving these requests, validates and processes the data, then updates the PostgreSQL database with the new task entries.

Task Coordination and Distribution

The **Coordinator** continuously polls the database for new tasks that are scheduled but not yet picked for execution. To manage concurrent access by multiple coordinators (in a scenario with multiple instances for scalability), the system uses PostgreSQL's `FOR UPDATE SKIP LOCKED` clause in SQL queries. This setup ensures that when multiple coordinators attempt to pick tasks simultaneously, they do not pick the same task. The clause allows a coordinator to skip rows (tasks) that are already locked by another coordinator, thus preventing conflicts and ensuring efficient distribution of tasks.

The Coordinator filters tasks that are due for execution within the next 30 seconds (`scheduled_at < (NOW() + INTERVAL '30 seconds')`), ensuring that the system is responsive and tasks are timely processed.

Task Execution

Once a task is picked by a Coordinator, it assigns the task to an available **Worker** node using gRPC, a high-performance, open-source universal RPC framework. The workers are responsible for the actual execution of the tasks. This interaction between the Coordinator and Workers is crucial for distributing the execution load and managing task processing across the system.

Status Updates and Monitoring

During the task execution process, Workers report the status of tasks back to the Coordinator. The Coordinator updates the task's status in the database based on these reports. The status updates include marking tasks as started, completed, or failed, along with the respective timestamps. This is handled by the `UpdateTaskStatus` function in the Coordinator, which updates the database records to reflect the current state of each task.

Docker Configuration

The system utilizes Docker to containerize and manage the different components. The `docker-compose` file defines four services: `postgres`, `scheduler`, `coordinator`, and `worker`. Each service is built from a custom Dockerfile, ensuring environment consistency and isolation.

Database Schema

The PostgreSQL database is equipped with a table `tasks` that records each task's command, scheduled execution time, and various timestamps reflecting different stages of task execution. This schema supports robust tracking and recovery mechanisms.

```
CREATE TABLE tasks (
    id UUID PRIMARY KEY DEFAULT uuid_generate_v4(),
    command TEXT NOT NULL,
    scheduled_at TIMESTAMP NOT NULL,
    picked_at TIMESTAMP,
    started_at TIMESTAMP,
    completed_at TIMESTAMP,
    failed_at TIMESTAMP
);
CREATE INDEX idx_tasks_scheduled_at ON tasks (scheduled_at);
```

Fault Tolerance in the Distributed Task Scheduler

The project addresses several challenges:

- **Task Duplication**: Ensured by the atomic task-picking mechanism in the coordinator.
- **Fault Tolerance**: Achieved through distributed architecture and worker heartbeats.
- **Scalability**: Facilitated by the stateless nature of the coordinator and workers, allowing horizontal scaling.

The distributed task scheduler system is designed with fault tolerance at its core, ensuring the system's reliability and availability even in the face of failures. This section delves into the strategies employed to mitigate potential failures, including worker node failures, task execution failures, and ensuring database integrity.

Worker Failure Management

The system employs a heartbeat mechanism to monitor the health of worker nodes. Worker nodes periodically send heartbeat messages to the Coordinator, indicating their operational status. This is facilitated by the `periodicHeartbeat` function, which is responsible for sending these messages at a predefined interval. The `sendHeartbeat` function constructs and sends these heartbeat messages, including the worker's ID and address.

```
func (w *WorkerServer) periodicHeartbeat() {
    w.wg.Add(1)
    defer w.wg.Done()

    ticker := time.NewTicker(w.heartbeatInterval)
    defer ticker.Stop()

    for {
        select {
        case <-ticker.C:
            if err := w.sendHeartbeat(); err != nil {
                log.Printf("Failed to send heartbeat: %v",
err)
                return
            }
        case <-w.ctx.Done():
            return
        }
    }
}
```

Upon receiving a heartbeat, the Coordinator updates the worker's status in its pool. If the worker is new, it is registered; otherwise, the Coordinator resets the heartbeat miss count to zero, indicating that the worker is active. This mechanism ensures that the Coordinator is aware of the active workers and can manage the worker pool accordingly.

Task Failure Management

The system's architecture inherently supports task failure management through its design. Tasks are scheduled and executed by worker nodes, which are responsible for their completion. If a worker fails to execute a task, the system's design ensures that the task is not lost but is instead reassigned to another available worker. This is facilitated by the Coordinator's ability to scan the database for tasks that have not been picked up and are due for execution. The `executeAllScheduledTasks` function, part of the Coordinator's database scanning process, identifies tasks that are scheduled but not yet picked up. It then submits these tasks to available workers for execution.

Database Interaction and Lock Management

The system uses PostgreSQL for storing task information and manages concurrent access to the database using locks. The Coordinator's database scanning process, as outlined in the `scanDatabase` and `executeAllScheduledTasks` functions, employs the `FOR UPDATE SKIP LOCKED` clause in SQL queries. This clause allows the Coordinator to select tasks that are due for execution and lock them for update, ensuring that multiple Coordinators do not pick up the same tasks. This mechanism prevents conflicts and ensures efficient task distribution, even in a distributed environment with multiple Coordinators.

```
func (s *CoordinatorServer) executeAllScheduledTasks() {
    ctx, cancel := context.WithTimeout(context.Background
(), 30*time.Second)
    defer cancel()

    tx, err := s.dbPool.Begin(ctx)
    if err != nil {
        log.Printf("Unable to start transaction: %v\n", er
r)
        return
    }

    defer func() {
        if err := tx.Rollback(ctx); err != nil && err.Error
() != "tx is closed" \\
        {
            log.Printf("ERROR: %#v", err)
            log.Printf("Failed to rollback transaction: %v
\n", err)
        }
    }()

    rows, err := tx.Query(ctx, `SELECT id, command FROM tas
ks WHERE \\
    scheduled_at < (NOW() + INTERVAL '30 seconds') AND pick
ed_at IS NULL \\
    ORDER BY scheduled_at FOR UPDATE SKIP LOCKED`)
    if err != nil {
        log.Printf("Error executing query: %v\n", err)
        return
    }
    defer rows.Close()

    var tasks []*pb.TaskRequest
    for rows.Next() {
        var id, command string
```

```go
        if err := rows.Scan(&id, &command); err != nil {
            log.Printf("Failed to scan row: %v\n", err)
            continue
        }

        tasks = append(tasks, &pb.TaskRequest{TaskId: id, D
ata: command})
    }

    if err := rows.Err(); err != nil {
        log.Printf("Error iterating rows: %v\n", err)
        return
    }

    for _, task := range tasks {
        if err := s.submitTaskToWorker(task); err != nil {
            log.Printf("Failed to submit task %s: %v\n", ta
sk.GetTaskId(), err)
            continue
        }

        if _, err := tx.Exec(ctx, `UPDATE tasks SET picked_
at = NOW() WHERE \\
         id = $1`, task.GetTaskId()); err != nil {
            log.Printf("Failed to update task %s: %v\n", ta
sk.GetTaskId(), err)
            continue
        }
    }

    if err := tx.Commit(ctx); err != nil {
        log.Printf("Failed to commit transaction: %v\n", er
r)
    }
}
```

The distributed task scheduler system's approach to fault tolerance is comprehensive, addressing potential failures at various levels of the system.

Through mechanisms like heartbeat messages for worker failure management, task reassignment for task failure management, and lock management for database interaction, the system ensures that it can continue to operate reliably and efficiently even in the face of failures. This robust fault tolerance strategy is crucial for maintaining the system's reliability and availability, making it suitable for use in critical computing environments. The system's design, which includes proactive measures for worker and task management, demonstrates a thoughtful approach to ensuring system resilience and minimising downtime.

Communication

The distributed task scheduler system employs a combination of HTTP/REST and gRPC for communication between its components. This section outlines the communication mechanisms, focusing on the API requests to the Scheduler, the interaction with the database, and the use of gRPC for communication between the Coordinator and Workers.

API Requests to the Scheduler

The Scheduler exposes two endpoints for API requests:

1. **POST /schedule**: This endpoint is used to schedule new tasks. The request body contains the command to be executed and the scheduled execution time in ISO 8601 format. The response includes the command, scheduled execution time, and a unique task ID.

2. **GET /status/**: This endpoint allows querying the status of a task using its task ID as a URL query parameter. The response includes detailed information about the task, such as its current status and timestamps for when it was picked, started, completed, or failed.

Database Interaction

Both the Scheduler and the Coordinator components interact with the PostgreSQL database to manage tasks. They utilize a connection pool (`dbPool`) to manage database connections efficiently. This pool is initialized with a context and a database connection string, enabling the execution of SQL queries and transactions.

For example, when a new task is scheduled, the Scheduler inserts the task into the database:

```
func (s *SchedulerServer) insertTaskIntoDB(ctx context.Context, task Task) (string, error) {
    sqlStatement := "INSERT INTO tasks (command, scheduled_at) VALUES ($1, $2) RETURNING id"
    var insertedId string
    err := s.dbPool.QueryRow(ctx, sqlStatement, task.Command, task.ScheduledAt.Time).Scan(&insertedId)
    return insertedId, err
}
```

The Coordinator, on the other hand, uses its `dbPool` to scan the database for tasks that are due for execution and to update the status of tasks as they are processed. This involves executing SQL queries to select tasks based on their scheduled time and updating their status in the database.

gRPC Communication

gRPC is used for efficient, low-latency communication between the Coordinator and Worker nodes. The gRPC contract defines several RPCs (Remote Procedure Calls) that facilitate this communication:

- **SubmitTask**: This RPC is used by the Coordinator to send tasks to Workers for execution. It includes the task ID and the command to be executed.
- **SendHeartbeat**: Workers periodically send heartbeat messages to the Coordinator to indicate their active status. This helps the Coordinator manage the worker pool and detect inactive workers.
- **UpdateTaskStatus**: Workers report the status of tasks back to the Coordinator using this RPC. It includes the task ID, the new status, and timestamps for when the task was started, completed, or failed.

The `pb.UnimplementedCoordinatorServiceServer` is a base type provided by the gRPC framework. It implements the `CoordinatorServiceServer` interface but leaves the RPC methods unimplemented. By embedding this type in the `CoordinatorServer` struct, the Coordinator can inherit the interface and then provide its own implementations for the RPC methods it supports. This approach allows for a clean separation of the gRPC service definition from its implementation, making the code more modular and easier to maintain.

Results

The distributed task scheduler system was successfully implemented and tested using Docker Compose, demonstrating its ability to handle task scheduling and execution across multiple nodes. The system's components, including the Scheduler, Coordinator, and Workers, were orchestrated using Docker, allowing for easy scaling and deployment.

```
docker-compose up --scale worker=3
```

This command scaled the worker service to three instances, simulating a distributed environment where tasks could be distributed across multiple nodes for execution.

Task Scheduling and Execution

To test the task scheduling functionality, a POST request was made to the Scheduler's `/schedule` endpoint with the following request body:

```
{
    "command" : "pwd",
    "scheduled_at" : "2024-04-20T2:05:53+05:30"
}
```

The Scheduler successfully processed this request and returned a response containing the command, the scheduled execution time (converted to a Unix timestamp), and a unique task ID:

```
{
    "command": "pwd",
    "scheduled_at": 1713558953,
    "task_id": "66ef60b4-f7b5-42d6-8745-ef9315b5739f"
}
```

This response confirmed that the task had been successfully scheduled and assigned a unique identifier.

Task Status Querying

The status of the scheduled task was then queried using the task ID returned in the previous response. A GET request was made to the Scheduler's `/status` endpoint with the task ID as a query parameter:

```
GET : http://localhost:8081/status?task_id=66ef60b4-f7b5-42d6
```

The response included detailed information about the task, including its current status and timestamps for when it was picked, started, and completed:

```
{
    "task_id": "66ef60b4-f7b5-42d6-8745-ef9315b5739f",
    "command": "pwd",
    "scheduled_at": "2024-04-19 20:35:53 +0000 UTC",
    "picked_at": "2024-04-20 08:34:16.408898 +0000 UTC",
    "started_at": "2024-04-20 08:34:16 +0000 UTC",
    "completed_at": "2024-04-20 08:34:21 +0000 UTC"
}
```

This response provided insights into the task's lifecycle, including the time required to complete the task, demonstrating the system's capability to track and report on task execution details.

The distributed task scheduler system demonstrated its effectiveness in scheduling and executing tasks across multiple nodes. The use of Docker Compose for orchestration facilitated easy scaling and deployment, while the system's API endpoints provided a straightforward interface for task scheduling and status querying. The detailed task status reporting enabled users to monitor task execution effectively, showcasing the system's robustness and reliability.

This project not only serves as a practical tool for distributed task scheduling but also as a valuable learning experience in distributed systems design and implementation. The system's design, which includes proactive measures for worker and task management, demonstrates a thoughtful approach to ensuring system resilience and minimising downtime.

Conclusion

The distributed task scheduler system, as developed and tested, showcases a robust and scalable approach to task scheduling and execution across multiple nodes. By leveraging Docker for containerisation, gRPC for efficient inter-service communication, and PostgreSQL for persistent state management, the system effectively addresses the challenges of reliability, fault tolerance, and scalability inherent in distributed systems.

The system's architecture, which includes a Scheduler for task submission, a Coordinator for task distribution and worker management, and Worker nodes for task execution, provides a flexible and efficient framework for managing distributed tasks. The use of Docker Compose for orchestration facilitated easy scaling and deployment, allowing for the dynamic adjustment of the system's capacity to meet varying operational demands.

The implementation of the system demonstrated its capability to handle task scheduling and execution, with tasks being distributed across multiple worker nodes based on their availability and the task's scheduled execution time. The system's API endpoints provided a user-friendly interface for task scheduling and status querying, enabling users to monitor and manage tasks effectively.

One of the key strengths of the system is its design for fault tolerance, which includes mechanisms for worker failure management, task failure management, and database interaction with lock management. These features ensure that the system can continue to operate reliably and efficiently even in the face of failures, thereby maintaining high availability and reliability.

Looking ahead, the system can be further enhanced by incorporating priority-based scheduling and deadline-sensitive scheduling. This can be achieved by adding a `priority` column to the SQL table for tasks, allowing tasks to be prioritised based on their importance. Similarly, a `deadline` column can be added to enable tasks to be scheduled with specific deadlines, ensuring that critical tasks are executed within the required timeframe. These enhancements would require minimal changes to the existing system architecture and could be implemented with relative ease, further expanding the system's capabilities and flexibility.

In conclusion, the distributed task scheduler system represents a significant advancement in distributed task management, offering a scalable, reliable, and efficient solution for scheduling and executing tasks across multiple nodes. The system's design and implementation provide a solid foundation for further development and enhancement, positioning it as a valuable tool for managing distributed tasks in various computing environments.

References

- Distributed Systems for Fun and Profit: A comprehensive guide to distributed systems, covering topics like consistency, availability, partition tolerance, and more.
- Designing Data-Intensive Applications: A book by Martin Kleppmann that provides insights into designing systems that store and process large amounts of data.
- gRPC Official Website: The official gRPC website, providing an overview of gRPC, its features, and how to use it.
- gRPC: Up and Running: A book by Bart Sigmund that introduces gRPC, covering its architecture, use cases, and how to implement gRPC services.
- Celery: Distributed Task Queue: While Celery is a Python-based task queue, its documentation provides valuable insights into distributed task scheduling and management.
- Apache Airflow: A platform to programmatically author, schedule, and monitor workflows, offering insights into task scheduling and workflow management.