**SCHOOL OF INFORMATION TECHNOLOGY AND ENGINEERING**

**ADDIS ABABA INSTITUTE OF TECHNOLOGY, ADDIS ABABA UNIVERSITY**

Topic –Distributed Task Scheduler in Golang

**Objective:**

Create a distributed task scheduler that:

1. Assigns tasks to multiple workers (simulating processors).
2. Handles worker failure by reassigning failed tasks.
3. Ensures balanced load distribution across workers.

**Step-by-step Instructions:**

1. **Define Tasks**: We will define a set of tasks that workers (processors) will process. Each task represents some work to be done (simulated with a sleep delay).
2. **Create Workers**: Workers are goroutines that will process tasks. We simulate them as processors that can fail randomly, and if they fail, the task will be reassigned to another worker.
3. **Task Reassignment on Failure**: If a worker fails during task execution, the task will be sent back to a fail queue for reassignment to another worker.
4. **Load Balancing**: Tasks will be distributed evenly to ensure no worker is overloaded.

**Implementation**  
  
**Step 1: Setup the Project**  
  
First, create a new directory and initialize a Go project.  
  
mkdir dist-task-scheduler  
cd dist-task-scheduler  
go mod init dist-task-scheduler  
  
**Step 2: Define Tasks and Workers**  
  
Here’s a Go program that defines tasks and workers, simulates failures, and redistributes tasks as necessary.  
  
package main  
  
import (  
    "fmt"  
    "math/rand"  
    "sync"  
    "time"  
)  
  
// Task represents a unit of work.  
type Task struct {  
    id   int  
    data string  
}  
  
// Worker function that processes tasks. If a worker fails, the task will be sent to failChan.  
func worker(id int, taskChan <-chan Task, wg \*sync.WaitGroup, failChan chan<- Task) {  
    defer wg.Done()  
  
    for task := range taskChan {  
        fmt.Printf("Worker %d started task %d: %s\n", id, task.id, task.data)  
  
        // Simulate random failure (30% chance of failure)  
        if rand.Float32() < 0.3 {  
            fmt.Printf("Worker %d failed on task %d\n", id, task.id)  
            failChan <- task // Send the failed task for reassignment  
            return  
        }  
  
        // Simulate task processing time  
        time.Sleep(time.Duration(rand.Intn(3)+1) \* time.Second)  
        fmt.Printf("Worker %d completed task %d\n", id, task.id)  
    }  
}  
  
func main() {  
    rand.Seed(time.Now().UnixNano())  
  
    // Define a set of tasks to be executed  
    tasks := []Task{  
        {id: 1, data: "Task 1"},  
        {id: 2, data: "Task 2"},  
        {id: 3, data: "Task 3"},  
        {id: 4, data: "Task 4"},  
        {id: 5, data: "Task 5"},  
    }  
  
    // Channels for task distribution and failure handling  
    taskChan := make(chan Task, len(tasks))  
    failChan := make(chan Task, len(tasks))  
  
    // WaitGroup to ensure all workers finish their tasks  
    var wg sync.WaitGroup  
  
    // Number of workers (simulating processors)  
    numWorkers := 3  
  
    // Start worker goroutines  
    for i := 1; i <= numWorkers; i++ {  
        wg.Add(1)  
        go worker(i, taskChan, &wg, failChan)  
    }  
  
    // Distribute tasks to workers  
    for \_, task := range tasks {  
        taskChan <- task  
    }  
    close(taskChan)  
  
    // Handle failed tasks by redistributing them  
    go func() {  
        for failedTask := range failChan {  
            fmt.Printf("Reassigning failed task %d\n", failedTask.id)  
            wg.Add(1)  
            go worker(rand.Intn(numWorkers)+1, taskChan, &wg, failChan)  
        }  
    }()  
  
    // Wait for all workers to finish  
    wg.Wait()  
    close(failChan)  
  
    fmt.Println("All tasks completed.")  
}  
  
**Step 3: Running the Program**  
  
Run the program using:  
  
go run main.go

**Explanation of the Code:**

1. **Task Distribution**:
   * We define a set of tasks ([]Task) that will be executed by the workers.
   * The taskChan channel is used to send tasks to the worker goroutines.
2. **Worker Goroutines**:
   * Workers are implemented as goroutines (go worker()), each processing tasks from taskChan.
   * Each worker has a random 30% chance of failure, which simulates real-world scenarios where a processor might fail.
3. **Failure Handling**:
   * When a worker fails (as simulated by a random failure), the task is sent to the failChan for reassignment.
   * Another worker picks up the task from failChan and tries to process it.
4. **Synchronization**:
   * A sync.WaitGroup ensures the program waits for all workers to finish before exiting.
5. **Load Balancing**:
   * Tasks are evenly distributed across the available workers using the taskChan.

**Assignment Exercise:**  
  
**Objective:**  
  
Modify the above task scheduler to introduce a mechanism and extend failure handling with a for failed tasks.  
  
**Instructions:**

* Implement a round-robin approach to ensure tasks are distributed in sequence to each worker.

            \* Use a counter to track which worker should get the next task.

* :Modify the failure-handling mechanism to include a maximum retry limit for each task. If a task fails more than a specified number of times, it should be logged as a failure and not retried further.
* :Add detailed logging for task assignments, completions, and failures.

**Hints**

* You can use a counter in the main function to distribute tasks to workers in a round-robin manner.
* Maintain a map to track how many times each task has been retried.

**What to submit**

* Submit the code file containing the program for each activity
* A screenshot showing the terminal output for each activity and the output showing the result of each activities