Detailed Report on MapReduce Implementation

Mohamad Saeed sedighi - 810100179
Mohamad Hosein Motaee - 810199493
Mahdi

April 18, 2025

1 Introduction

The report is structured to include a detailed examination of each file's code, an explanation of its functionality, and an analysis of the test output from running the `Test` script, as depicted in the attached image.

2 Code Analysis

2.1 coordinator.go

```go
package mr

import (
    "log"
    "net"
    "net/http"
    "net/rpc"
    "os"
    "sync"
    "time"
)

type Coordinator struct {
    mu          sync.Mutex
    mapTasks    []Task
    reduceTasks []Task
    mapDone     bool
    reduceDone  bool
```

```
    nReduce      int
    nMap         int
    tasks        map[int]*TaskTracker
}

type TaskTracker struct {
    Status    TaskState
    WorkerId  string
    StartTime time.Time
}

func (c *Coordinator) server() {
    rpc.Register(c)
    rpc.HandleHTTP()
    sockname := coordinatorSock()
    os.Remove(sockname)
    l, e := net.Listen("unix", sockname)
    if e != nil {
        log.Fatal("listen error:", e)
    }
    go http.Serve(l, nil)
}

func (c *Coordinator) monitor() {
    for {
        c.mu.Lock()
        if c.Done() {
            c.mu.Unlock()
            return
        }

        now := time.Now()
        for _, tracker := range c.tasks {
            if tracker.Status == InProgress &&
   now.Sub(tracker.StartTime) > 10*time.Second {
                tracker.Status = Pending
                tracker.WorkerId = ""
                tracker.StartTime = time.Time{}
            }
        }
        c.mu.Unlock()
        time.Sleep(time.Second)
    }
}

func (c *Coordinator) Done() bool {
    return c.mapDone && c.reduceDone
}

```

```
67 func MakeCoordinator(files []string, nReduce int)
       *Coordinator {
68     c := Coordinator{
69         mapTasks:    make([]Task, len(files)),
70         reduceTasks: make([]Task, nReduce),
71         tasks:       map[int]*TaskTracker{},
72         nReduce:     nReduce,
73         nMap:        len(files),
74     }
75
76     for i, file := range files {
77         c.mapTasks[i] = Task{Map, i, file, nReduce,
       len(files)}
78         c.tasks[i] = &TaskTracker{Pending, "", time.Time{}}
79     }
80
81     for i := 0; i < nReduce; i++ {
82         c.reduceTasks[i] = Task{Reduce, i, "", nReduce,
       len(files)}
83         c.tasks[len(files)+i] = &TaskTracker{Pending, "",
       time.Time{}}
84     }
85
86     go c.monitor()
87     c.server()
88     return &c
89 }
90
91 func (c *Coordinator) AssignTask(args *TaskArgs, reply
       *TaskReply) error {
92     c.mu.Lock()
93     defer c.mu.Unlock()
94
95     if !c.mapDone {
96         for i, task := range c.mapTasks {
97             if c.tasks[i].Status == Pending {
98                 c.tasks[i].Status = InProgress
99                 c.tasks[i].WorkerId = args.WorkerId
100                c.tasks[i].StartTime = time.Now()
101                reply.Task = task
102                return nil
103            }
104        }
105        c.mapDone = true
106        for i := 0; i < c.nMap; i++ {
107            if c.tasks[i].Status != Completed {
108                c.mapDone = false
109                break
110            }
```

```
111         }
112     }
113
114     if c.mapDone && !c.reduceDone {
115         for i, task := range c.reduceTasks {
116             taskId := c.nMap + i
117             if c.tasks[taskId].Status == Pending {
118                 c.tasks[taskId].Status = InProgress
119                 c.tasks[taskId].WorkerId = args.WorkerId
120                 c.tasks[taskId].StartTime = time.Now()
121                 reply.Task = task
122                 return nil
123             }
124         }
125         c.reduceDone = true
126         for i := c.nMap; i < c.nMap+c.nReduce; i++ {
127             if c.tasks[i].Status != Completed {
128                 c.reduceDone = false
129                 break
130             }
131         }
132     }
133
134     reply.Task = Task{Type: None}
135     return nil
136 }
137
138 func (c *Coordinator) TaskDone(args *ReportArgs, reply
    *ReportReply) error {
139     c.mu.Lock()
140     defer c.mu.Unlock()
141
142     taskId := args.TaskNum
143     if args.TaskType == Reduce {
144         taskId += c.nMap
145     }
146
147     if c.tasks[taskId].WorkerId != args.WorkerId {
148         reply.Success = false
149         return nil
150     }
151
152     c.tasks[taskId].Status = Completed
153     reply.Success = true
154     return nil
155 }
```

2.1.1 Explanation

The `coordinator.go` file is the backbone of the MapReduce system, implementing the `Coordinator` struct and its associated methods. The coordinator is responsible for orchestrating the entire MapReduce workflow, including task assignment, progress tracking, and fault tolerance.

- `Structs`: - `Task`: Represents a MapReduce task, which could be a map or reduce task. Fields include: - `Type`: Indicates whether the task is a map or reduce task. - `Num`: The task number. - `Filename`: The input file for map tasks (empty for reduce tasks). - `NumReduce` and `NumMap`: The number of reduce and map tasks. - `TaskArgs`: Contains the worker's ID. - `TaskReply`: Contains the task assigned by the coordinator. - `ReportArgs`: Contains the worker's ID, task number, and task type (map or reduce). - `ReportReply`: Indicates whether the task completion report was successful.

- `Methods`: - `server()`: Initializes an RPC server over a Unix socket (generated by `coordinatorSock()`). It registers the coordinator for RPC calls, removes any existing socket file, and starts an HTTP server in a goroutine to handle worker requests. - `monitor()`: Runs continuously in a goroutine to detect stalled tasks. It locks the mutex, checks each task in progress, and resets tasks to `Pending` if they exceed a 10-second timeout, enhancing fault tolerance by reassigning failed tasks. - `Done()`: Returns `true` when both map and reduce phases are complete, allowing the system to terminate gracefully. - `MakeCoordinator()`: Constructs a new coordinator instance. It initializes map tasks from input files, creates reduce tasks, sets up task trackers, and launches the monitor and server goroutines. - `AssignTask()`: An RPC method that assigns tasks to workers. It prioritizes map tasks until all are completed (`mapDone` becomes `true`), then assigns reduce tasks. If no tasks are available, it returns a `None` task type. The method updates task status and tracks worker assignment. - `TaskDone()`: An RPC method that marks a task as completed. It validates the worker's ID and updates the task status to `Completed`. If the report is valid, it returns `true`, signaling success.

2.2 rpc.go

```
1 package mr
2
3 import (
4     "net/rpc"
5 )
6
7 type Task struct {
8     Type       TaskType
```

```
    Num        int
    Filename   string
    NumReduce  int
    NumMap     int
}

type TaskArgs struct {
    WorkerId string
}

type TaskReply struct {
    Task Task
}

type ReportArgs struct {
    WorkerId string
    TaskNum  int
    TaskType TaskType
}

type ReportReply struct {
    Success bool
}

type TaskType int

const (
    Map TaskType = iota
    Reduce
    None
)

func call(rpcname string, args interface{}, reply
   interface{}) bool {
    client, err := rpc.DialHTTP("unix", coordinatorSock())
    if err != nil {
        return false
    }
    defer client.Close()

    err = client.Call(rpcname, args, reply)
    if err != nil {
        return false
    }
    return true
}
```

2.2.1 Explanation

The `rpc.go` file defines the data structures used for remote procedure calls (RPCs) between the coordinator and the workers. It also contains the `call()` function to facilitate communication with the coordinator.

- `Structs`: - `Task`: Represents a MapReduce task, which could be a map or reduce task. Fields include: - `Type`: Indicates whether the task is a map or reduce task. - `Num`: The task number. - `Filename`: The input file for map tasks (empty for reduce tasks). - `NumReduce` and `NumMap`: The number of reduce and map tasks. - `TaskArgs`: Contains the worker's ID. - `TaskReply`: Contains the task assigned by the coordinator. - `ReportArgs`: Contains the worker's ID, task number, and task type (map or reduce). - `ReportReply`: Indicates whether the task completion report was successful. - `TaskType`: An enumeration defining the three possible types of tasks: Map, Reduce, and None.

- `Function`: - `call()`: This function is used by workers to send RPC requests to the coordinator. It opens a connection to the coordinator via a Unix socket and sends the RPC call. If the call is successful, it returns `true`, otherwise `false`.

2.3 worker.go

```
1  package mr
2
3  import (
4      "fmt"
5      "log"
6      "os"
7      "time"
8  )
9
10 func worker() {
11     for {
12         var args TaskArgs
13         var reply TaskReply
14         if !call("Coordinator.AssignTask", &args, &reply) {
15             log.Fatal("Call to Coordinator failed!")
16         }
17
18         task := reply.Task
19         if task.Type == None {
20             return
21         }
22
23         if task.Type == Map {
```

```
            // Perform Map Task
            err := doMapTask(task)
            if err != nil {
                log.Printf("Map task failed: %v", err)
                continue
            }
        } else if task.Type == Reduce {
            // Perform Reduce Task
            err := doReduceTask(task)
            if err != nil {
                log.Printf("Reduce task failed: %v", err)
                continue
            }
        }

        var reportArgs ReportArgs
        var reportReply ReportReply
        reportArgs.WorkerId = args.WorkerId
        reportArgs.TaskNum = task.Num
        reportArgs.TaskType = task.Type

        if !call("Coordinator.TaskDone", &reportArgs,
    &reportReply) {
            log.Fatal("Call to Coordinator failed!")
        }

        if !reportReply.Success {
            log.Printf("Failed to report task completion for
    task %d", task.Num)
        }
     }
}
```

2.3.1 Explanation

The `worker.go` file implements the worker's behavior. It continuously requests tasks from the coordinator, processes them, and reports the completion back to the coordinator.

- `worker()`: The worker function enters an infinite loop where it: - Calls the `Coordinator.AssignTask` RPC to request a task. - If the task is `None`, it terminates. - Depending on the task type (map or reduce), it calls the respective function (`doMapTask()` or `doReduceTask()`) to perform the task. - After completing the task, it calls `Coordinator.TaskDone` to report the completion of the task.

3 Test Output and Analysis

The output generated by running the test script `Test` demonstrates the proper functioning of the MapReduce system, as each worker completes its assigned task (either map or reduce) and reports the completion to the coordinator.

```
saeid@saeid:~/Desktop/Distributed/1/Project 1/Project 1/Distributed_Systems_CA1$ cd src/main
saeid@saeid:~/Desktop/Distributed/1/Project 1/Project 1/Distributed_Systems_CA1/src/main$ bash test-mr.sh
*** Starting wc test.
--- wc test: PASS
2025/04/18 03:27:44 dialing:dial unix /var/tmp/5840-mr-1000: connect: connection refused
2025/04/18 03:27:44 dialing:dial unix /var/tmp/5840-mr-1000: connect: connection refused
2025/04/18 03:27:44 dialing:dial unix /var/tmp/5840-mr-1000: connect: connection refused
*** Starting indexer test.
2025/04/18 03:27:48 dialing:dial unix /var/tmp/5840-mr-1000: connect: connection refused
2025/04/18 03:27:48 dialing:dial unix /var/tmp/5840-mr-1000: connect: connection refused
--- indexer test: PASS
*** Starting map parallelism test.
2025/04/18 03:27:55 dialing:dial unix /var/tmp/5840-mr-1000: connect: connection refused
2025/04/18 03:27:55 dialing:dial unix /var/tmp/5840-mr-1000: connect: connection refused
--- map parallelism test: PASS
*** Starting reduce parallelism test.
2025/04/18 03:28:03 dialing:dial unix /var/tmp/5840-mr-1000: connect: connection refused
2025/04/18 03:28:03 dialing:dial unix /var/tmp/5840-mr-1000: connect: connection refused
--- reduce parallelism test: PASS
*** Starting job count test.
2025/04/18 03:28:21 dialing:dial unix /var/tmp/5840-mr-1000: connect: connection refused
2025/04/18 03:28:21 dialing:dial unix /var/tmp/5840-mr-1000: connect: connection refused
2025/04/18 03:28:21 dialing:dial unix /var/tmp/5840-mr-1000: connect: connection refused
--- job count test: PASS
2025/04/18 03:28:22 dialing:dial unix /var/tmp/5840-mr-1000: connect: connection refused
*** Starting early exit test.
2025/04/18 03:28:29 dialing:dial unix /var/tmp/5840-mr-1000: connect: connection refused
2025/04/18 03:28:29 dialing:dial unix /var/tmp/5840-mr-1000: connect: connection refused
2025/04/18 03:28:29 dialing:dial unix /var/tmp/5840-mr-1000: connect: connection refused
--- early exit test: PASS
*** Starting crash test.
2025/04/18 03:29:11 dialing:dial unix /var/tmp/5840-mr-1000: connect: connection refused
2025/04/18 03:29:12 dialing:dial unix /var/tmp/5840-mr-1000: connect: connection refused
2025/04/18 03:29:12 dialing:dial unix /var/tmp/5840-mr-1000: connect: connection refused
--- crash test: PASS
*** PASSED ALL TESTS
saeid@saeid:~/Desktop/Distributed/1/Project 1/Project 1/Distributed_Systems_CA1/src/main$
```

Figure 1: Test Output