CSC4160 Final Project: Cloud-Enabled Distributed MapReduce System Implementation

Midterm Report

Name & Student ID:

Boshi Xu (122040075)

Danyang Chen (123090018)

1. Introduction

1.1 Background and Motivation

The rapid growth of data in various domains has led to an increasing demand for efficient processing and analysis methods. Traditional single-node systems struggle to handle the scale and complexity of modern datasets. To address this, distributed computing models like MapReduce have emerged as powerful solutions. Originally introduced by Google, MapReduce simplifies large-scale data processing by dividing tasks into smaller, manageable subtasks that can be executed in parallel across multiple nodes. This approach has proven effective in industries ranging from e-commerce to scientific research[1].

In recent years, cloud computing has further revolutionized distributed systems. By leveraging cloud platforms such as AWS, organizations can dynamically allocate resources, ensuring scalability, fault tolerance, and cost efficiency. These advancements make it feasible to deploy robust distributed systems that can process massive datasets while maintaining high availability and resilience. This project integrates these cutting-edge technologies to develop a cloud-enabled distributed MapReduce system, combining the strengths of the MapReduce model with the flexibility and scalability of cloud infrastructure.

1.2 Objectives

The primary goal of this project is to implement a functional, cloud-based distributed MapReduce system. This system aims to achieve the following objectives:

- **Develop a Coordinator and Worker Framework**:
  Implement a distributed MapReduce system with one coordinator and multiple workers running in parallel. Workers communicate with the coordinator via RPC, requesting tasks, processing input files, writing outputs, and looping to request new tasks. The coordinator monitors worker progress and reassigns tasks if a worker fails to complete them within ten seconds, ensuring fault tolerance and efficient task execution.

- **Leverage Cloud Resources**:
  Utilize AWS services like EC2 for computation, S3 for data storage, and Auto Scaling for dynamic resource allocation based on workload demands.

- **Ensure Fault Tolerance and High Availability**:
  Design the system to handle worker failures gracefully, ensuring continuous operation without manual

intervention.

- **Optimize for Scalability and Performance**:
  Demonstrate the system's ability to efficiently process large datasets by scaling horizontally across multiple nodes.

- **Enhance Security (Optional)**:
  Implement data security measures, including encryption and controlled access using AWS IAM roles and policies.

By achieving these goals, the project will showcase how modern cloud computing technologies can enhance traditional distributed systems, offering a scalable, resilient, and efficient solution for large-scale data processing.

---

2. Project Roadmap

2.1 Requirement Analysis and Design [Week 1-2]

- **Requirement Definition**: Analyze project requirements and establish functional and performance objectives.
- **System Design**: Design the system architecture, focusing on the interaction between the coordinator and workers.
- **Technology Selection**: Choose suitable cloud services, such as AWS EC2 and S3, for system deployment.

2.2 Implementation and Testing [Week 3-7]

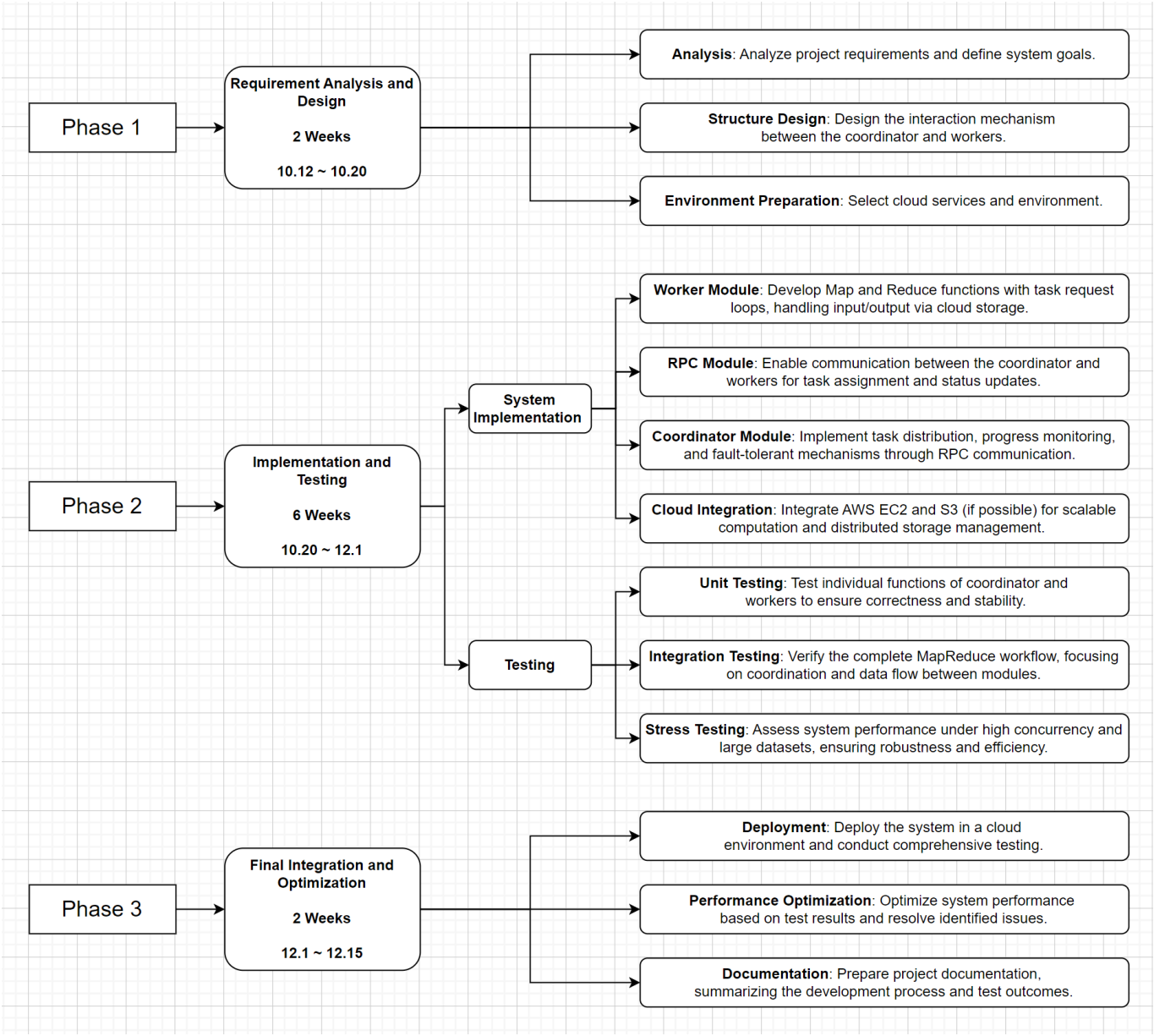
- **System Implementation**:
  - **Worker Implementation**: Implement the worker module to perform Map and Reduce operations, handling data processing tasks.
  - **Coordinator Implementation**: Develop the coordinator module for task assignment, progress tracking, and fault-tolerance mechanisms.
  - **Cloud Integration**: Integrate AWS EC2 for computation and S3 for distributed storage to enable a scalable system.

- **Testing**:
  - **Unit Testing**: Test individual functions of the coordinator and worker modules to ensure accuracy and reliability.
  - **Integration Testing**: Validate the complete MapReduce workflow, focusing on seamless communication and data processing.
  - **Stress Testing**: Assess system performance under high concurrency and large datasets to ensure robustness and efficiency.

2.3 Final Integration and Optimization [Week 8-10]

- **Deployment**: Deploy the system in a cloud environment and conduct comprehensive testing.

- **Performance Optimization**: Optimize system performance based on test results and resolve identified issues.
- **Documentation**: Prepare project documentation, summarizing the development process and test outcomes.



3. Preliminary Results

3.1 Overview

The MapReduce framework follows the foundational concepts outlined in Google's paper [1]. We will construct a distributed MapReduce system in a local environment first, where tasks are divided between a central coordinator and multiple workers.

As of now, the **worker module** and the **RPC framework** have been implemented. These components form the backbone of the distributed system, allowing workers to process tasks and communicate with the coordinator. While the coordinator module is still under development, the implemented parts have been tested independently using mock data and placeholder methods.

3.2 Worker and RPC Implementation

The worker and RPC frameworks have been developed to handle task execution and communication in a MapReduce environment:

---

1. **Worker Module**

   The worker module manages the execution of Map and Reduce tasks assigned by the coordinator. Its modular design allows workers to dynamically request and process tasks, ensuring efficient task handling.

- **Map Task Execution**:
  - Reads input files and processes their contents using the user-defined `mapf` function.
  - Partitions intermediate key-value pairs into `nReduce` buckets, writing them to intermediate files.
- **Reduce Task Execution**:
  - Aggregates intermediate results from Map tasks by reading key-value pairs.
  - Applies the user-defined `reducef` function to compute the final output and writes the results to output files.

Core Functions:

- **Worker Entry Point**:

```
func Worker(mapf func(string, string) []KeyValue, reducef func(string,
[]string) string)
```

This function acts as the main loop for the worker, handling task requests and executing Map or Reduce tasks dynamically based on assignments from the coordinator.

- **Task Processing**:

```
func handleMapTask(taskID int, inputFile string, nReduce int, mapf
func(string, string) []KeyValue) bool
func handleReduceTask(taskID int, nReduce int, reducef func(string,
[]string) string) bool
```

These functions implement the core logic for processing tasks. They handle input files, generate intermediate files for Map tasks, and create output files for Reduce tasks.

---

2. **RPC Framework**

   The RPC framework facilitates communication between workers and the coordinator, ensuring task assignments and status updates are efficiently handled.

- **Task Requests**:
  - Workers use the `CallForTask` method to request tasks from the coordinator. This ensures dynamic allocation of tasks based on the coordinator's queue.

  Function Signature:

  ```
  func CallForTask() *CoordinatorResponse
  ```

  This function sends an RPC request to the coordinator to obtain task details. It returns a `CoordinatorResponse` containing the task type, task ID, input file (if any), and the number of Reduce tasks (`nReduce`).

- **Task Status Reporting**:
  - After completing or failing a task, workers use the `CallForReportStatus` method to notify the coordinator of the task status.

  Function Signature:

  ```
  func CallForReportStatus(successType MsgType, taskID int)
  ```

  This function sends an RPC call to the coordinator, reporting the success or failure of a task. This enables the coordinator to track progress and reassign tasks if necessary.

Core RPC Function:

- **RPC Communication**:

  ```
  func call(rpcname string, args interface{}, reply interface{}) bool
  ```

  This generic function handles all RPC communication between workers and the coordinator. It sends a request (`args`) to a specified RPC method (`rpcname`) and receives a response (`reply`).

Summary

The worker and RPC frameworks have been successfully implemented, enabling workers to:

1. Dynamically request tasks from the coordinator.
2. Process tasks independently using user-defined `mapf` and `reducef` functions.
3. Report task completion or failure to the coordinator.

These functionalities ensure seamless communication and efficient task execution within the MapReduce framework.

3.3 Independent Testing Results

The worker and RPC frameworks were tested independently using the word count application plugin provided by the MIT Course 6.5840 Lab [2].

1. **Map Task Execution**:
    - The `Map` function splits the content of an input file (`pg-1.txt`) into words, emitting each word as a key with a value of `"1"`.
    - Intermediate files (`mr-1-0`, `mr-1-1`, `mr-1-2`) were successfully created, with key-value pairs partitioned based on the hash of the key.
2. **Reduce Task Simulation**:
    - Mock intermediate files from the Map phase were aggregated using the `Reduce` function to compute word counts.
    - Output files (`mr-out-0`) were successfully generated, containing results in the expected format (e.g., `word1 5`).

These results validate the functionality of the worker module for word count tasks, confirming its ability to handle both Map and Reduce phases independently.

3.4 Challenges

1. **Coordinator Module**:
    - Without a functional coordinator, dynamic task assignment and fault tolerance could not be tested.
2. **Fault Tolerance**:
    - Mechanisms for handling worker crashes and task failures remain untested in the current setup.
3. **Cloud Deployment**:
    - Ensure the system operates seamlessly on AWS EC2 instances, leveraging cloud resources for distributed processing.
4. **Performance Validation**:
    - Conduct thorough testing on the cloud to validate system scalability, fault tolerance, and overall performance under real-world conditions.

3.5 Next Steps

1. **Coordinator Development**:
    - Complete the coordinator module to handle dynamic task distribution and fault recovery.
2. **Comprehensive Integration Testing**:

- Test end-to-end functionality, including worker-coordinator communication and task reassignment.

3. **Scalability and Fault Tolerance**:
   - Introduce multiple workers to test the system's behavior under high concurrency.
   - Simulate worker crashes to validate fault tolerance mechanisms.

4. **Documentation**:
   - Record test cases, results, and performance metrics for inclusion in the final report.

This progress demonstrates the successful implementation of the worker and RPC modules, laying a solid foundation for the subsequent development of the coordinator and the completion of the distributed MapReduce system.

---

4. Reference

1. Dean, J., & Ghemawat, S. (2004). MapReduce: Simplified Data Processing on Large Clusters. *OSDI '04: 6th Symposium on Operating Systems Design and Implementation*, USENIX Association.
2. *MIT Graduate Course 6.5840: Distributed Systems (Spring 2024) - Lab: MapReduce*. Retrieved from https://pdos.csail.mit.edu/6.824/labs/lab-mr.html.