CSC4160 Final Report: Cloud-Enabled Distributed MapReduce System Implementation

> **Team Members:**
>
> - Danyang Chen (123090018)
> - Boshi Xu (122040075)

> **Video Demo:** Bilibili Video (Please turn on the Subtitles when watching)

> **Code Repository Link:** Github Repository

1. Motivation and Background

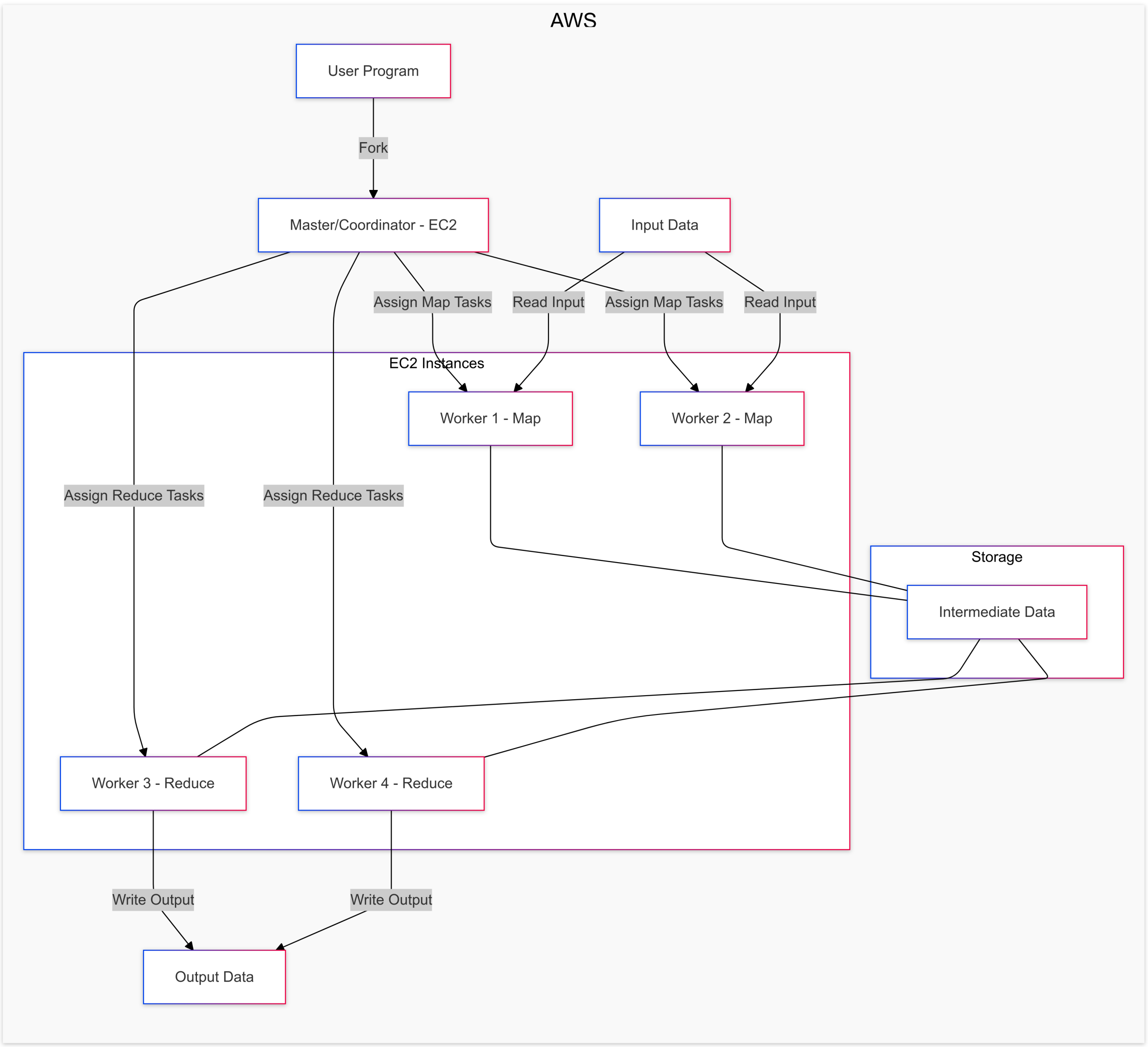
In this project, we have implemented a cloud-enabled distributed MapReduce system, inspired by the MapReduce framework introduced by Google. MapReduce is a programming model used for processing and generating large datasets by dividing tasks into smaller subtasks, which are then executed in parallel across multiple nodes. The original concept of MapReduce was detailed in a paper by Jeffrey Dean and Sanjay Ghemawat, presented at **OSDI '04: 6th Symposium on Operating Systems Design and Implementation, USENIX Association**, which highlighted its efficiency in handling large-scale data processing by utilizing a master-worker architecture [1].

We focuses on building a cloud-enabled distributed system, leveraging modern cloud computing technologies to achieve scalability, fault tolerance, and high availability. Cloud platforms like **AWS** provide the essential infrastructure needed to run large-scale MapReduce operations, utilizing cloud services such as **Amazon EC2** and **Amazon S3**.

---

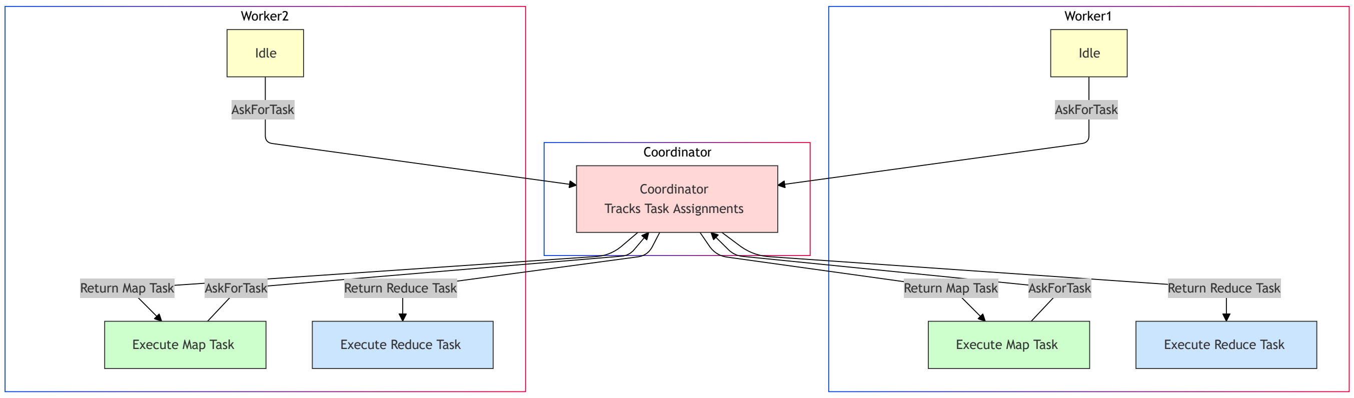
2. System Overview

Overall Architecture



master-worker architecture:

1. **Coordinator (Master)**:
    - Distributes Map and Reduce tasks to Workers.
    - Tracks task status and progress.
    - Handles fault tolerance by reassigning failed or timed-out tasks.
2. **Workers**:
    - Execute assigned Map or Reduce tasks.
    - Report the task status (success or failure) back to the Coordinator.
    - Handle intermediate data by creating partitioned files during Map tasks and processing them during Reduce tasks.



Communication Design

The Coordinator and Workers communicate through **Remote Procedure Calls (RPC)** over Unix domain sockets. The system uses structured messages to ensure efficient and clear communication.

MapReduce Deployment

The MapReduce system is deployed on **AWS cloud infrastructure** to implement the expected cloud functions. `EC2` instances are used to run the coordinator and multiple workers to achieve distributed task execution. `S3` is used to store inputs, demonstrating the integration of cloud servers and cloud storage. This deployment demonstrates the scalability, reliability, and applicability of the system in a real cloud environment.

---

3. Design Details

3.1 RPC Communication

Communication between the Coordinator and Workers is implemented using structured message types:

```go
type WorkerRequest struct {
    Type   MsgType // The type of message being sent
    TaskID int     // Task ID for completion or failure notifications
}

type CoordinatorResponse struct {
    Type      MsgType // The type of message being sent
    TaskID    int     // ID of the assigned task
    InputFile string  // Input file for Map tasks
    NumReduce int     // Number of Reduce tasks
}
```

3.2 Worker Implementation

The `Worker` function orchestrates task requests, execution, and reporting:

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

- The Worker loops, requesting tasks from the Coordinator and executing them based on the type of task assigned (`Map` or `Reduce`).
- Task execution involves reading input files, applying the appropriate user-defined function (`mapf` or `reducef`), and handling intermediate or final data outputs.
- The Worker reports task success or failure to the Coordinator.

Map Task Execution:

- Reads input files, applies the `mapf` function to generate key-value pairs, and partitions intermediate results into buckets.
- Writes each bucket to a temporary file and atomically renames it to ensure consistency.

Reduce Task Execution:

- Reads intermediate files, groups values by key, applies the `reducef` function, and writes results to a temporary output file, which is atomically renamed to finalize output.

Status Reporting:

- Reports success or failure of tasks back to the Coordinator to update task states and trigger reassignment if needed.

3.3 Coordinator Implementation

The Coordinator manages task assignments, monitors progress, and handles task reassignments:

```
type Coordinator struct {
    mu          sync.Mutex
    mapTasks    map[int]*MapTaskInfo
    reduceTasks map[int]*ReduceTaskInfo
    nReduce     int
    mapDone     bool
    reduceDone  bool
    taskTimeout time.Duration
}
```

Task Assignment:

```
func (c *Coordinator) AskForTask(args *WorkerRequest, reply *CoordinatorResponse)
error
```

- Assigns idle tasks to Workers or reassigns timed-out tasks for both Map and Reduce stages.
- If all tasks are completed, it sends a termination signal (`CoordinatorEnd`) to Workers.

Timeout Monitoring:

```go
func (c *Coordinator) monitorTimeouts()
```

- Periodically checks for timed-out tasks and resets their statuses to `idle` for reassignment.

Task Status Updates:

```go
func (c *Coordinator) NoticeResult(args *WorkerRequest, reply *struct{}) error
```

- Updates task statuses (e.g., `finished`, `failed`) based on Worker reports.
- Checks for overall task completion.

3.4 Deployment and Implementation on AWS EC2 and S3

1. Compute Resources - Amazon EC2

- The **Coordinator** and multiple **Workers** were deployed on **EC2 instances** (m5.large, Ubuntu 24.04 LTS) to efficiently execute distributed Map and Reduce tasks.
- Worker processes were dynamically managed, simulating the elasticity of cloud resources.

2. Data Storage - Amazon S3

- Input data was stored in **Amazon S3** for high availability and durability.
- A **Python script** fetched input files from S3 to the local directories of EC2 instances, enabling seamless data management for large-scale processing.

4. Achievements and Evaluation

1. Successfully implemented Map and Reduce task execution.
2. Fault-tolerant task reallocation ensured reliability in case of Worker failures.
3. Successfully deployed the system on AWS cloud infrastructure, integrating `EC2` for computation and `S3` for data management, demonstrating scalability and real-world applicability.
4. Successfully passed all the test scripts provided by the **MIT 6.5840 Distributed Systems** course, verifying the correctness and robustness of the system [2].

References

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.