Distributed MapReduce Implementation Using gRPC

1. Overview

This project implements a simplified distributed MapReduce framework using gRPC to coordinate tasks between a master process and multiple worker processes (mappers and reducers). The system supports two operations: **Word Count** and **Inverted Index**. The solution is structured into separate components for the master, mappers, and reducers with clearly defined gRPC services to facilitate communication between them.

2. Architecture Overview

Components

- **Master (Client):**
    - Splits the input data into chunks and assigns them to mapper workers.
    - Coordinates the map and reduce phases by invoking gRPC calls on the workers.
    - Collects intermediate data paths from mappers and sends the corresponding partitions to reducers.
    - Terminates worker processes when their tasks are complete.
    - Accepts configuration via the command line (CLI) for operation selection (Word Count or Inverted Index), number of mappers/reducers, ports for each worker, and input/output directories.
- **Mapper Workers:**
    - Run as independent gRPC servers.
    - Each mapper receives a subset of input files, applies the mapping function (either for Word Count or Inverted Index), and writes intermediate key/value pairs into multiple partition files.
    - The partitioning is based on a simple hash (for example, the word length modulo the number of reducers) to evenly distribute keys across reducer workers.
- **Reducer Workers:**
    - Run as independent gRPC servers.
    - Each reducer receives one partition (i.e., a set of intermediate files corresponding to its partition number) from all mappers.

    - The reducer aggregates the intermediate data (e.g., sums counts for Word Count or aggregates file identifiers for Inverted Index) and writes the final output.

3. Supported Operations

Word Count

- **Mapper Operation:**
    - Reads the assigned input files line by line.
    - Splits each line into words.
    - For each word, writes an intermediate key/value pair in the format `<word> 1` into one of the partition files.
    - The partition is determined using a simple hash (e.g., `len(word) % num_reducers`).
- **Reducer Operation:**
    - Reads the partition files received from mappers.
    - Groups key/value pairs by word.
    - Sums the counts for each word.
    - Writes the final output in the format `<word> <total_count>` into the output file.

Inverted Index

- **Mapper Operation:**
    - Reads the assigned input files.
    - For each line, splits the line into words.
    - For each word, writes an intermediate key/value pair in the format `<word> <file_id>` (where `file_id` corresponds to the identifier of the input file) into the appropriate partition file.
- **Reducer Operation:**
    - Reads the partition files from mappers.
    - Groups key/value pairs by word.
    - Aggregates file IDs (removing duplicates) for each word.
    - Writes the final output in the format `<word> <list_of_file_ids>` into the output file.

4. gRPC Services and RPC Usage

The use of gRPC provides a structured and efficient mechanism for communication between the master and worker nodes. Two main services are defined using Protocol Buffers (.proto files):

Mapper Service

- **Proto File:** `mapper_service.proto`
- **Service Name:** `MapWorker`
- **RPC Method:** `PerformMap`

Request Message (`MapReq`):

- `task_type`: Specifies the operation (1 for Word Count, 2 for Inverted Index).
- `data_dir`: The directory containing input files.
- `file_names`: List of input files assigned to the mapper.
- `file_ids`: Identifiers corresponding to the input files (used in Inverted Index).
- `reduce_count`: The total number of reducers; used to create the appropriate number of partition files.

Response Message (`MapResp`):

- `temp_dir`: The directory where the mapper writes its intermediate (partitioned) outputs.
- `status`: Enum indicating whether the mapping operation was successful (`OK`) or encountered an error (`ERROR`).

Reducer Service

- **Proto File:** `reducer_service.proto`
- **Service Name:** `ReduceWorker`
- **RPC Method:** `PerformReduce`

Request Message (`ReduceReq`):

- `task_type`: Specifies the operation (1 for Word Count, 2 for Inverted Index).
- `part_files`: List of partition file paths collected from all mapper workers corresponding to this reducer's partition.
- `output_dir`: Directory where the final output is to be stored.

Response Message (`ReduceResp`):

- `final_output`: The final output file path (or directory) containing the reduced results.
- `status`: Enum indicating the success (`OK`) or failure (`ERROR`) of the reduce operation.

5. Input via CLI

When executing the master process, the user is prompted to provide several configuration parameters via the command line:

- **Operation Selection:**
  - The user selects the operation (Word Count by entering `1` or Inverted Index by entering `2`).
- **Number of Mappers and Reducers:**
  - The user specifies how many mapper and reducer processes to launch.
- **Ports for Mappers and Reducers:**
  - The user enters the ports (space-separated) that will be used by each mapper and reducer process.
- **Input and Output Directories:**
  - The master uses relative paths (for example, `../dataset` for input and `../output` for output) which are resolved relative to the master's working directory.

This CLI-driven approach allows flexible configuration without hard-coding system parameters.

6. Data Flow and Partitioning

- **Input Splitting:**
  The master process scans the input directory (e.g., a folder named `dataset`) and splits the files among the available mapper workers. The split logic ensures that if the number of input files is less than the number of mappers, some mappers may remain idle, and the system gracefully handles such cases.

- **Intermediate Output Generation:**
  Each mapper writes its output into multiple partition files (one per reducer) under a temporary directory (for example, `temp_results/worker_mapper_1`). Each file contains one line per key/value pair (formatted as `<key> <value>`), ensuring that the reducers receive consistent, partitioned data.

- **Reduction and Final Output:**
  The master aggregates the intermediate directories and constructs partition-specific file paths for each reducer. Reducers read their assigned partition files, perform aggregation (summing counts for Word Count or aggregating file IDs for Inverted Index), and write the final output to the designated output directory.

7. Communication Flow

1. **Map Phase:**
    - The master uses a thread pool to concurrently call the `PerformMap` RPC on each mapper.
    - Mappers process their assigned files and return the location of their intermediate output.
    - The master collects these intermediate directories.
2. **Reduce Phase:**
    - The master determines the correct partition files for each reducer.
    - It concurrently invokes the `PerformReduce` RPC on each reducer.
    - Reducers process the intermediate data and generate the final output.
3. **Worker Termination:**
   After the map and reduce phases complete, the master explicitly terminates the worker processes, ensuring that all worker nodes exit.

8. Test Case

Below is an example test case illustrating both operations (Word Count and Inverted Index) with 3 mappers and 3 reducers.

Test Case 1: Word Count Operation

- **Configuration:**
    - **Number of Mappers:** 3
    - **Number of Reducers:** 3
    - **Operation Selected:** 1 (Word Count)
- **Dataset Files:**

dataset/input1.txt:

```
Bye , bye , bye ,
Bye , bye , don't want to be a fool for you ,
Just another player in your game for two ,
You may hate me , but it ain't no lie , baby , bye , bye
```

dataset/input2.txt:

```
In this world , it’s just us
You know it’s not the same as it was
As it was , as it was
You know it’s not the same
```

dataset/input3.txt:

```
I'm in love with the shape of you
We push and pull like a magnet do
Although my heart is falling too
I'm in love with your body
```

- **Intermediate Mapper Output:**
  - **worker_mapper_1:**
    - *part_0:*
      ```
      bye 1 bye 1 bye 1 bye 1 bye 1 for 1 you 1 player 1 for 1 two 1 you 1 may 1 but 1 lie 1 bye 1 bye 1
      ```
    - *part_1:*
      ```
      , 1 , 1 , 1 , 1 , 1 want 1 a 1 fool 1 , 1 just 1 another 1 your 1 game 1 , 1 hate 1 , 1 , 1 baby 1 , 1 , 1
      ```
    - *part_2:*
      ```
      don't 1 to 1 be 1 in 1 me 1 it 1 ain't 1 no 1
      ```
  - **worker_mapper_2:**
    - *part_0:*
      ```
      you 1 not 1 the 1 was 1 was 1 was 1 you 1 not 1 the 1
      ```
    - *part_1:*
      ```
      this 1 , 1 it’s 1 just 1 know 1 it’s 1 same 1 , 1 know 1 it’s 1 same 1
      ```
    - *part_2:*
      ```
      in 1 world 1 us 1 as 1 it 1 as 1 it 1 as 1 it 1
      ```
  - **worker_mapper_3:**
    - *part_0:*
      ```
      i'm 1 the 1 you 1 and 1 magnet 1 too 1 i'm 1
      ```
    - *part_1:*
      ```
      love 1 with 1 push 1 pull 1 like 1 a 1 falling 1 love 1 with 1 your 1 body 1
      ```
    - *part_2:*
      ```
      in 1 shape 1 of 1 we 1 do 1 although 1 my 1 heart 1 is 1 in 1
      ```
- **Reducer Final Output:**
  - **reducer_1 (from partition 0):**
    ```
    i'm 2 the 3 you 5 and 1 magnet 1 too 1 not 2 was 3 bye 7 for 2 player 1 two 1 may 1 but 1 lie 1
    ```

- **reducer_2 (from partition 1):**
  `love 2 with 2 push 1 pull 1 like 1 a 2 falling 1 your 2 body 1 this 1 , 13 it’s 3 just 2 know 2 same 2 want 1 fool 1 another 1 game 1 hate 1 baby 1`
- **reducer_3 (from partition 2):**
  `in 4 shape 1 of 1 we 1 do 1 although 1 my 1 heart 1 is 1 world 1 us 1 as 3 it 4 don't 1 to 1 be 1 me 1 ain't 1 no 1`

Test Case 2: Inverted Index Operation

- **Configuration:**
  - **Number of Mappers:** 3
  - **Number of Reducers:** 3
  - **Operation Selected:** 2 (Inverted Index)
- **Intermediate Mapper Output:**
  - **worker_mapper_1:**
    - *part_0:*
      `bye 0 bye 0 bye 0 bye 0 bye 0 for 0 you 0 player 0 for 0 two 0 you 0 may 0 but 0 lie 0 bye 0 bye 0`
    - *part_1:*
      `, 0 , 0 , 0 , 0 , 0 want 0 a 0 fool 0 , 0 just 0 another 0 your 0 game 0 , 0 hate 0 , 0 , 0 baby 0 , 0 , 0`
    - *part_2:*
      `don't 0 to 0 be 0 in 0 me 0 it 0 ain't 0 no 0`
  - **worker_mapper_2:**
    - *part_0:*
      `it’s 1 you 1 it’s 1 not 1 the 1 was 1 was 1 was 1 you 1 it’s 1 not 1 the 1`
    - *part_1:*
      `this 1 , 1 just 1 know 1 same 1 , 1 know 1 same 1`
    - *part_2:*
      `in 1 world 1 us 1 as 1 it 1 as 1 it 1 as 1 it 1`
  - **worker_mapper_3:**
    - *part_0:*
      `i'm 2 the 2 you 2 and 2 magnet 2 too 2 i'm 2`

        - *part_1:*
          ```
          love 2 with 2 push 2 pull 2 like 2 a 2 falling 2 love 2 with 2 your 2 body 2
          ```
        - *part_2:*
          ```
          in 2 shape 2 of 2 we 2 do 2 although 2 my 2 heart 2 is 2 in 2
          ```
- **Reducer Final Output:**
    - **reducer_1 (from partition 0):**
      ```
      i'm 2 the 1, 2 you 0, 1, 2 and 2 magnet 2 too 2 it’s 1 not 1 was 1 bye 0 for 0 player 0 two 0 may 0 but 0 lie 0
      ```
    - **reducer_2 (from partition 1):**
      ```
      love 2 with 2 push 2 pull 2 like 2 a 0, 2 falling 2 your 0, 2 body 2 this 1 , 0, 1 just 0, 1 know 1 same 1 want 0 fool 0 another 0 game 0 hate 0 baby 0
      ```
    - **reducer_3 (from partition 2):**
      ```
      in 0, 1, 2 shape 2 of 2 we 2 do 2 although 2 my 2 heart 2 is 2 world 1 us 1 as 1 it 0, 1 don't 0 to 0 be 0 me 0 ain't 0 no 0
      ```

---

9. Conclusion

This report documents the design and implementation of a distributed MapReduce system using gRPC. The solution meets the assignment requirements by:

- Defining clear gRPC services for mapping and reducing tasks.
- Using a master process to coordinate data distribution and task scheduling.
- Implementing worker nodes that generate partitioned intermediate outputs and then aggregate these outputs to produce the final results.
- Supporting both the Word Count and Inverted Index operations.
- Allowing configuration via the CLI for operation type, number of mappers/reducers, ports, and directories.
- Including a comprehensive test case that demonstrates the system's behavior for both operations with 3 mappers and 3 reducers, with partition outputs formatted with two-space separation.

This modular, scalable architecture can be extended to support additional functionalities if needed, and it provides a solid foundation for understanding distributed processing using gRPC.