DESIGN DOCUMENT

CompSci 532: Project 1

Chirag Uday Kamath *Divya Maiya* *Neha Prakash*

I. Program Design

The system implements a version of MapReduce running on a single server, while mimicking the behavior of a distributed system. The system allows users to write custom Map Reduce functions that can be run by the system in a fault-tolerant and parallel manner. The system works on a single device utilizing the cores for multiprocessing. The files are considered as different partitions of the data.

The system works in a very similar manner to Hadoop. Processes can communicate only by explicitly sending and receiving messages through IPC or sockets (with the exception of intermediate data, which is exchanged through the intermediate files mentioned previously) over a network similar to the Hadoop ecosystem. The details about partitioning functions, intermediate storage of key-value pairs, etc. directly follow from the Google MapReduce paper.
The user provides the input data directory and every file within the directory is considered as the partition of the data that will be consumed by the "Map". The user also provides an output directory location where the output of every Reducer will be dumped. Along with this, the user also writes "Map" and "Reduce" functions that are run by the system on the
provided data.
Along with the input and output location, the user also provides the number of mappers/reducers (N). Fault Tolerance Logic is implemented to handle any number of Mappers or Reducers becoming unavailable at random times.

Applications implemented

Word Count: Given an input file containing a number of sentences/words, populate an output file containing the number of occurrences of each word in the input.

- Input: The input files consist of text files containing different paragraphs of text.
- Output: The output files give the count of each word in all the files.

Mutual Friends: For each pair of people in a social network, count the mutual friends

- Input: A list of pairs consisting of mapping between a person and his/her friends. For example: A: B, C, D shows that a person A has friends B, C, D.
- Output: The output is a pair mapping from a pair of users to their mutual friends. For example, (A, B) > (C, D) implies that C, D are the mutual friends of (A, B).

Application Logs: For a given list of logs, find the number of error codes such as 404, 500 and 304.

- Input: Application log files

- Pair containing the error code and the number of times it has occurred.

Server Requests: Given a list of IP addresses that sent requests to a particular server, populate an output file that only consists of IP addresses that made over a certain threshold (10 in our case) number of requests

II. How it works

User-Defined Functions

When started, your executable will act as a master and launch *N mapper* processes, which will each scan a separate partition of the rows of the input file. The mappers will invoke the user-defined map function on each row in their partition and write the output of the map function to an intermediate file on the local file system. Java Reflection API is used for this task. The user of the system implements a Client interface, particularly, it implements two functions, namely “Map” and “Reduce”. These functions need to be declared public and the path of this implementation is provided within the user-provided config file.

The Java file containing the implementation is initially compiled by the system. Following compilation, the system transmits the file's path to each worker node (mapper or reducer) using socket connection. Every worker retrieves the Client instance by loading the class at the specified path. Every Mapper worker can use the "Map" function and every Reducer worker can call the "Reduce" function that is user-written using the Reflection API.

Parallelism

This task is carried out using the Java ProcessBuilder API. Because communication between two worker nodes in any generic Map-Reduce system occurs over the network rather than over a shared memory stack, multiprocessor systems are used.

The degree of parallelism is controlled by "N," one of the entries in the user-supplied config file. For the Map task, "N" workers are established, each receiving various data partitions. At the end of the map phase N*N intermediate files are created.

N different reduce workers work on the intermediate files such that each intermediate file is going to be consumed by one and only one reducer. One reducer could take one or more than one intermediate file from each mapper.

Initially, the master begins all of Mapper's "N" processes at the same time. After that, it waits for all of the mapper processes to finish before launching "N" new Reducer processes.

Interprocess communication

Socket Programming is used to communicate between the Master Process and the various Mapper and Reducer processes. Arguments and file locations are passed between these processes using Data Input and Data Output Streams. Between the Mapper and Reducer, there is no connection. The Master Process is used for all communication. On PORT 6666, the Master Process starts a Server

Socket that waits for incoming requests. Every client (either a Mapper or a Reducer worker) connects to this socket and uses a TCP connection to communicate with the server. At the Master node, a separate thread is maintained for parallel communication.
Once a Mapper or a Reducer completes its task successfully, it ends the process and returns an exit code of 0. Hence, if a non-zero exit code is received, fault tolerance logic needs to be invoked.

Fault Tolerance

The Master includes a list of processes (either Mapper or Reducer) with a process ID for each of them. When the Master detects that a process is dead (as indicated by a non-zero exitcode), it executes fault handling code, which generates a new process with the same process id as the dead one. Every process broadcasts its process id to the Server at the start so that the Server knows which partition should be assigned to that process. The matching process creates the required output on its assigned partition.

Intermediate Key-Value Pairs

The master splits the input partitions into N sections and each Mapper receives and operates on one section. For example, N = 4, meaniEach mapper takes 3 partitions and outputs R=16 intermediate files. ng that you have 4 mappers and 4 reducers. The input file can be partitioned into M=12 partitions. The intermediate (key, value) of a partition goes to one of the N files. This is determined using a Partitioning function. The default partitioning function that we use is - hash(key) mod N as indicated in the Map-Reduce paper. Each reducer takes 4 intermediate files from each mapper and in total 4*N= 16 intermediate files. Each reducer outputs one output file; since you have N=4 reducer, you will have 4 output files.

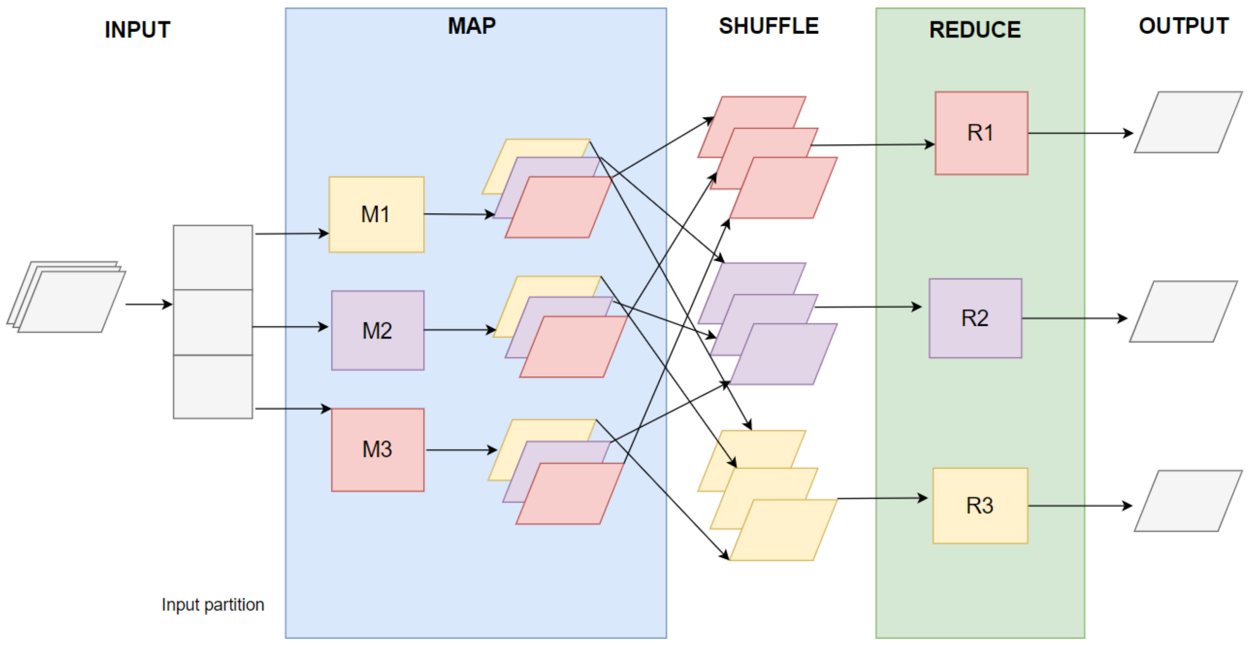

Fig: Mapreduce architecture

III. Design Tradeoffs

Firstly, the design considers that multiple files will be provided as input in the input directory. Hence, each file can be treated as a separate partition. Without this consideration, the entire file will have to be read by a process for it to be split into partitions, which should be avoided. Hence, in our implementation the master does not read the input files, it directly splits them among the workers.

Secondly, for the purpose of demonstration of fault tolerance, our script decides whether the mapper or the reducer will become unavailable at a random time. The testing script also destroys one worker only. To execute this, steps have been provided in the IV. How to run a section.

IV. How to Run

Run the following commands in `map-reduce-teamcdn-2` directory:

Windows

```
$ ./runnerScriptWindows.sh
```

Linux

```
$ chmod +x runnerScriptMac.sh
$ ./runnerScriptMac.sh
```

The above shell scripts run the four examples that are explained in the I. Program Design part of the code. Further, more information regarding the examples are provided in the README.md file

Fault tolerance

Run the following commands in `map-reduce-teamcdn-2` directory:

Windows

```
$ ./runnerScriptWindowsFaultTolerance.sh
```

Linux

```
$ chmod +x runnerScriptMacFaultTolerance.sh
$ ./runnerScriptMacFaultTolerance.sh
```