Design Document : MapReduce

Team : Jiye Choi, Sora Ryu, Chiehmin Chung

This program is to implement MapReduce on a single server. Even though it works on a single server, it mimics the behavior of a distributed implementation, which multi processes communicate by using Remote Method Invocation. A user will pass a 'configuration file' including an application name, an input txt file location, a directory location for output files, and the number of workers, N.

Master-Slave Architecture

This MapReduce system is implemented in the Master-Slave architecture. The master launches multiple slaves, and the slaves work as Mappers or Reducers. Before the master starts its job, it splits the user input file into partitions based on N. Then, the master creates N processes to invoke Mapper Servers by executing MapperServers.jar file. Each Mapper server has its own port number for registry. Therefore, when the processes execute the MapperServers jar file, they must pass the unique port number as an argument. The port number starts from 1099 to 1099+N for Mapper Server. To call the mapper server's methods, the master creates N threads. Each Thread sends requests to a different mapper server, and gets intermediate file locations. Each file has a region number at the end of its name. Before the master passes Reducers the locations returned by Mapper Servers, the master kills current running processes running on Mapper Servers. As the master launches mappers, the master creates N processes, and the processes execute Reducer Servers jar file. Like the mapper server, each reducer server has its own port number for the registry. Thus, the processes should pass the port numbers as an argument as well. For Reducer Server, the port number starts from 1098 to 1098-N. Finally, Master creates N threads, and each thread sends Reducer Server requests with intermediate file locations. Each reducer is responsible for all intermediate files with a different region number . For instance, reducer 1 gets all locations of intermediate files with region number 1. After all of the reducers complete their jobs, the master kills the processes and terminates the program.

In a Mapper side, a mapper gets a location of a partition, which are indices of a start line and an end line, from the master. Then, it splits the partition and invokes a user-defined map function on each row of the splitted partition. The mapper collects all outputs from the user-define function, and, then, it splits these outputs into N regions by using a simple hash-based partitioning algorithm. In order to make each reducer have something to work on, we just simply set R = N in this program. Then, it writes them on intermediate files named 'intermediate_file[the mapper pid]_[region number]'. Finally, it returns the file locations to the master. Overall, since there are N mappers, eventually N * N intermediate files exist.

In a Reducer side, a reducer gets multiple intermediate file locations from the master. Then, it reads and stores data from all assigned intermediate files, and shuffles and sorts the files by key. On each key, it invokes a user-defined reduce function. Finally, the reducer writes the outputs in an output file. Overall, each reducer reads N intermediate files which contain a specific region number. Thus, all intermediate files are covered, and not even duplicated. Also, since there are N reducers, eventually N output files exist.

This program is also able to tolerate worker’s faults. The master detects a worker's failures. When one of the worker servers (either mapper or reducer) fails, the server throws 'RemoteException'. Then, the exception is caught by the master server, and the master relaunches a new process. This new process worker works on the part that the faulty worker was responsible for. To test this fault tolerance,

we intentionally introduce a failure on a random mapper and reducer after we launch mappers/reducers. It happens in the library code (mapreduce class).

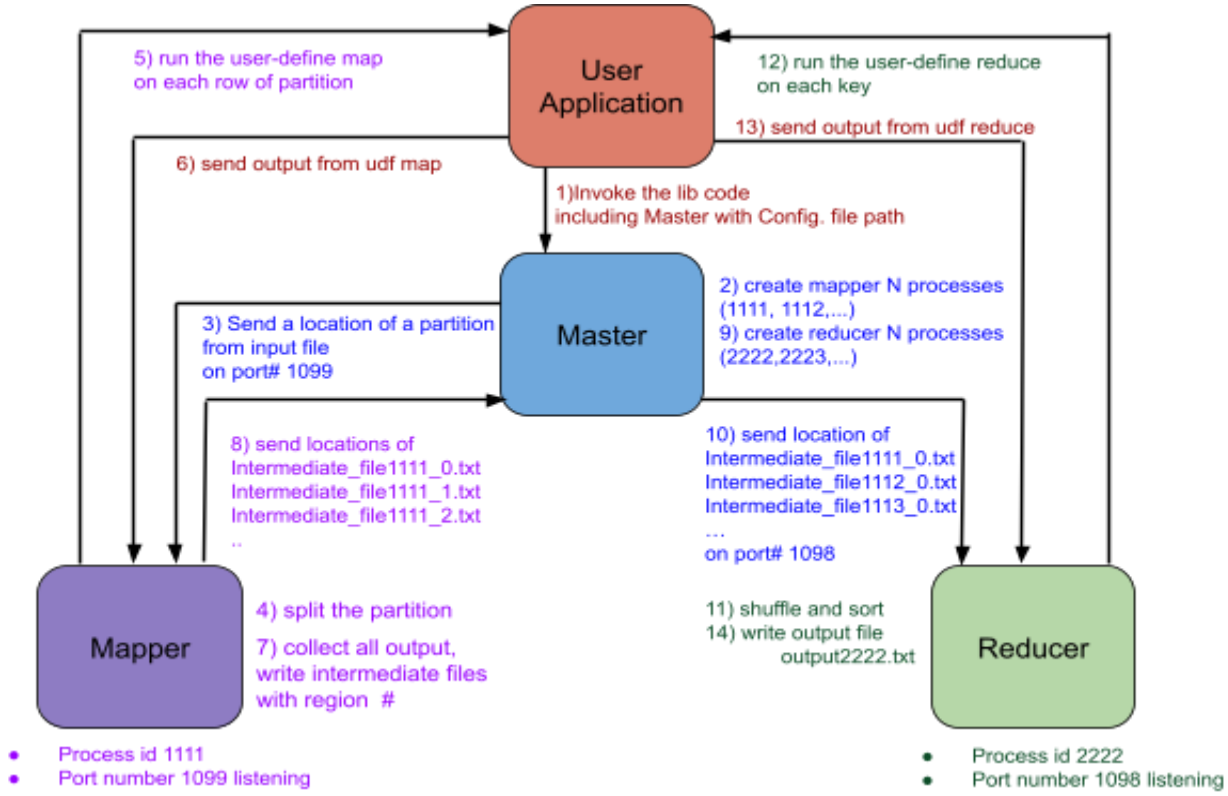

Figure1. MapReduce Architecture and Process

Design Tradeoff

As we mentioned earlier, this library code runs on a single server unlike other MapReduce libraries. But we tried to implement the behavior of distributed systems with RMI. Since the master could be blocked while the worker server is working, we make the master create multiple threads and send requests to multiple worker servers in parallel. Another consideration was when we implemented multiple workers, we discussed if it would be better to make a master worker server to share one port number, and several slave worker servers to communicate with the master through the master worker server. However, we thought the architecture of the code would be more complicated, and it would be more difficult to implement. Thus, for simplicity, we decided to use unique port numbers for each worker server.

In addition, we had to make many decisions while implementing the program. There were so many options on development, such as how to split input data and how to partition the intermediate files. Through our numerous team discussions, we developed our own input split function and intermediate partition function. For splitting the input, it evenly splits into N numbers of partitions based on the line units by using the quotient and remainder. Each worker will get basic partition size, and if the remainder exists, each will be additionally assigned to some of the workers. Thus, if the input file has a total of 6 lines and N = 4, it will be splitted into 2,2,1,1 lines respectively. In the end of the function, it returns indices of start line and end line, not transferring the data.

For the intermediate file partitioning function, we used the djb2 hash function algorithm for two main reasons. First of all, if they have the same key, it at least puts them into the same intermediate file, and this can help the reducer to easily apply the reduce function in the future. Also, the djb2 hash function is one of the commonly used hash functions and it has excellent distribution with simple implementation.

How to use

First of all, a user should make sure that the user's application class file, MapperServers.jar, ReducerServers.jar are in the 'src' folder. For instance, MapReduce/src/WordCount.java. The jar files are compiled by JDK 15. Thus, the user needs JDK 15 or newer version of it. In order to use this mapreduce library code properly, please follow the following instructions below.

1) the user's application class should implement 'Mapper' and 'Reducer' interfaces to implement map, reduce functions. We recommend you to make a new folder which has exactly the same name as the application class file, and save configuration file and input file there.
2) In the user's configuration file (config.properties), the user must provide the user application's name, input file path, output file path, and the number of workers you wish. Lastly, the user application should call the MapReduce.main passing the configuration file path. For example, MapReduce.main(new String[] {"application's name/config.properties"});
3) Run these command line in terminal
    - javac [your application].java (e.g., javac WordCount.java)
    - java [your application] (e.g., java WordCount)

Test cases

We also provide three test applications; Word Count, Count the same length words and Count Vowels. Each application will be executed in three different behaviors; with a single process, with multiple processes, and with multiple processes and one fault. The brief explanation of each test application is as follows.

- Word Count : to count how many times each word has occurred in an input file.
- Count the same length words: To count the same length of words in an input file.
- Count Vowels: To count the same number of vowels of words in an input file.

There are three test cases ; a single process, multiple processes, and multiple processes with fault tolerance. To test all cases at the same time, please run the single test script (test.sh) in the root folder. If the user wishes to test the cases separately, run each test script in test_scripts. All test cases are in the test_cases folder.

1) In order to run all test cases run the command line in the root folder.
    - bash test.sh
2) In the test_scripts folder, there are three separate test case scripts.
    - bash test_single.sh // to test mapreduce with a single process
    - bash test_multi.sh // to test mapreduce with multiple processes
    - bash test_final.sh // to test mapreduce with multiple processes and a fault tolerance