Distributed System 15-640 Lab3

Student Name: Yu-Hsin Kuo, Hao-Ping Ho
Andrew ID: yuhsink, haopingh

Configuration

In this project, we build a system of MapReduce engine. This system consists of a "Resource Allocator" as the system master, and many working nodes as Mappers and Reducers. At the beginning, the system is initialized manually based on the two system configuration files. The first file "master" defines the IP address and port of the Resource Allocator so that user can submit a job to the Resource Allocation, and get the allocated Mappers and Reducers for this job. The second file "slaves" defines the whole Mappers and Reducers which will run in the systems. The Resource Allocator will also check the status of every Mappers and Reducers. Therefore, it can allocate available (alive and idle) Mappers and Reducers to each MapReduce job requests. At the end of every job, Resource Allocator will release the resources used by the previous job.

As for the user part, when a user wants to run MapReduce job on our system, he/she has to first define the Task Configuration file "taskconfig". In the file, the requested amount of Mappers and Reducers, and the function names of Map and Reduce are defined. These information will be used by the main program of a MapReduce task so that every mapper and reducer can execute the functions defined by users.

Running a Map-Reduce Process

Here, we give a brief description of how to run the our map-reduce facility. The detail architecture and design will be described in other sections. To instantiate the program, log in to the clusters which you want it to be a mapper node or a reducer node and then run the "runMapper.sh" or "runReducer.sh". What the scripts do is for each cluster node, they will instantiate 3 threads to be as either mapper or reducer. Once you are done with all the map-reduce tasks, do remember to run "clean.sh" to release all the resources.

After the mapper and reducer programs start running, you then run the "ResourceAllocator" program to control the whole system. The Resource Allocator tracks the status of all mappers and reducers, and allocates these resources based on the availability information. Then, by executing the "Master" with the task configuration file, Map/Reduce java files, and input file, the "Master" program will communicate with the Resource Allocator to get the IPs and ports of available Mappers and Reducers. Finally, the "Master" program will send split files to Mappers, and keep tracking the whole MapReduce task until it completes, or fails.

Design of Mapper Nodes and Client Nodes

Mapper Nodes

The workflow of our mapper nodes are: (1) create all necessary connections, (2) download the chunk from the master node, (3) download the java file from the master node, (4) execute the mapper function and (5)

send the results to the reducer nodes.
(1) Create all necessary connections: As soon as the program starts, we establish the connection between mappers and the master node.
(2) download the chunk from the master node: Here we download the chunk from the master node with the portion number of this file and the job name. The reason of using job name is that we can guarantee each file is unique in the local directory. Because if you keep running different program, the same working node might download the file with the same portion number as previous and with the job name, you won't overwrite existing files.
(3) download the java file from the master node: This is similar as (2) except here we need to use the exact name of the original java file.
(4) execute the mapper function: We use reflection to instantiate the object and call the map function. Depending on how many reducers are uses, the "Output" object will generate same amount number of intermediate output locally.
(5) send the file to reducers: From the previous step, each output file corresponding to one reducer and therefore we can know how to send those files.

Reducer Nodes

The workflwo of our reducer nodes are very similar to mapper nodes and they are: (1) download the java file from the master node, (2) execute and (3) send the results to the master node.
(1)download the java file from the master node: same as in mapper nodes.
(2)execute: The reducer will wait until it received the message from the master node saying that it can start executing the reduce program. That's because only the master node knows if all the mappers finish sending all the necessary files to the reducers.
(3)send the results to the master node: The master will keep asking the status of each reducer node. Once our reducers nodes finish executing, they will be in idle status and by that time the transmission of the result starts.

Concurrent Use

Our system can be used by many users. Users can launch a MapReduce task from any participant, and the "Master" main program will do all communications with Resource Allocators, mappers and reducers. Two users will not use the same mapper or reducer at the same time. That is, a working thread on a working node will not be shared by two tasks at the same time because our Resource Allocator ensures the resource not being reallocated based on the synchronized data structure of availability information. The final result of a MapReduce job will be located in the directory which a user launch this job. The whole job process will not overlap another job unless the user unsafely develop their MapReduce functions.

Scheduling

Our resource allocator will keep track of the status of all mapper/reducer nodes. Every time the tasks are done, they will update the current status of the node with the resource allocator. To avoid overusing some nodes, we adopted a round-robin scheduling strategy which is simple but fair for all the working nodes. However, in the future, we might have other strategies such as using different weights for different nodes. For nodes that are very robust, we can assign higher weight (because they are unlikely to crash!) so that we can still want to wait for it to be available instead of using less robust nodes.

Failure and Recovery

Here, we classified the failure into two main categories, reducer failure and mapper failure. The strategies adopted here are similariy to each other. Let's start with mapper failure.

Mapper failure

For the mappers we are going to use, we assign a back-up mapper for each one of them. For example, if this task requires X mappers, we will use 2X mappers and half of them we call "original mappers" and the other half we call them "back-up mappers". To start, the master first notify all "original mappers" to start execution. At the same time, all the back-up mappers are listening to the port connecting to the master. By listening to this port, they will know if they need to do any operation. Once any of the original mapper crashes during execution, the master will catch the error and then find out the back up mapper and notify them to execute. If the original mapper execute without failures or errors the master will still find out the corresponding back-up mapper and notify them to reset and become idle again.

Reducer failure

Similar to mapper failure, we also use two times the number of the required reducers. What we are doing here is that the mappers will generate two copies of the output and send one to the original reducer and send the other one to the corresponding back-up reducers. Then, each one of the reducers wait for the master's notification. If the original reducer doesn't crash, then it will send the result to the master and the master will ask the back-up reducer to reset. Otherwise if it crashes, the master will ask the result from the back up reducer.

To test this, in our implementation we will force the thread to sleep for 10 seconds so that you have time to kill any process you want and see how our system handles the failure.

Map-Reduce Library and File I/O Library

To use our mapper and reducer interface, users need to implements those interfaces and implement required function (e.g. reduce and map function). The functions are defines as follows:

public void map (String value, Output output)

public void reduce (String key, List <String> values, Output output)

For the map function, basically, the value represents one line in the original text. The "Output" type provides **write (String key, String value)** function that can write the key-value pair to some output file. The "Output" class automatically handles which key-value pairs go to which file so that the users don't need to worry about that. They can just write any key-value pair they want.

For the reduce function, the values is a list of string containing the values related to that specific key. Therefore, users can just iterate through this list to get all the values of the same key and perform any computation they want. The "Output" object provided here again allows users to write outputs to some files.

To see how to use those functions properly, you can refer to the examples codes, e.g. WordCountMapper.java, WordCountReducer.java, RemoveNoiseMapper.java and RemoveNoiseReducer.java.

Examples

The examples given here are: (1) word count and (2) remove noise.

Word count

The mapper function will split the input text based on space and writes the word and number 1 as key-value pair to the output. The reducer aggregates the results and get the total count for each word. The final result is **resultWordCount.txt**.

Remove noise

The mapper function does the same thing as the word count program. The reducer will have a specified filter string and if the key contains that string it will be removed at the final answer otherwise it will exist in the output. The final result is **resultRemoveNoise.txt** and we set the filter string as "a" so that you can see the output file does not have words containing "a".

Run our program

Step1: Modify all the necessary configuration files as in Fig 1, Fig 2 and Fig 3

```
Master 1
ghc53.ghc.andrew.cmu.edu 5566
```

Figure 1: Master Config

```
Mapper 9
ghc51.ghc.andrew.cmu.edu 8000 8080
ghc51.ghc.andrew.cmu.edu 8001 8081
ghc51.ghc.andrew.cmu.edu 8002 8082
ghc50.ghc.andrew.cmu.edu 8000 8080
ghc50.ghc.andrew.cmu.edu 8001 8081
ghc50.ghc.andrew.cmu.edu 8002 8082
ghc49.ghc.andrew.cmu.edu 8000 8080
ghc49.ghc.andrew.cmu.edu 8001 8081
ghc49.ghc.andrew.cmu.edu 8002 8082
Reducer 6
ghc52.ghc.andrew.cmu.edu 7000 7010 7020
ghc52.ghc.andrew.cmu.edu 7001 7011 7021
ghc52.ghc.andrew.cmu.edu 7002 7012 7022
ghc53.ghc.andrew.cmu.edu 7000 7010 7020
ghc53.ghc.andrew.cmu.edu 7001 7011 7021
ghc53.ghc.andrew.cmu.edu 7002 7012 7022
```

Figure 2: Slave Config

```
mapper 5
reducer 2
mapper_class TestMapper
mapper_function map
reducer_class TestReducer
reducer_function reduce
```

Figure 3: Task Config

Step2: Log in to the clusters and execute "runMapper.sh" or "runReducer.sh" which depends on how you assign for each cluster.
Step3: Go to the clustere where you assign it as the resource allocater and run "java Resource Allocator"
Step4: Log in to any node and run "java Master taskconfig ¡Mapper.java¿ ¡Reducer.java¿ ¡input file¿"
Step6: Then you can see the final result, "results.txt", will be generated in your local directory