Lab 4: Map Reduce on Fault-tolerant Distributed Filesystem

Getting started

Before starting this lab, please back up all of your prior labs' solutions.

```
$ cd cse-lab
$ git commit -a -m "upload lab3-sol"
# Then, pull this lab from the repo:

$ git pull
# Next, switch to the lab4 branch:

$ git checkout lab4
# Notice: lab4 is based on lab3.

# Please merge with branch lab3, and solve the conflicts.

$ git merge lab3
# After merging the conflicts, you should be able to compile the new project successfully:

$ chmod -R o+w `pwd`

$ sudo docker run -it --rm --privileged --cap-add=ALL -v `pwd`:/home/stu/cse-lab
lqyuan980413/cselab_env:2022lab4
/bin/bash
$ cd cse-lab
$ make clean && make
```

(Reference: MIT 6.824 Distributed Systems)

In this lab, you are asked to build a MapReduce framework on top of your Distributed Filesystem implemented in Lab1-3.

You will implement a worker process that calls Map and Reduce functions and handles reading and writing files, and a coordinator process that hands out tasks to workers and copes with failed workers.

You can refer to the MapReduce paper for more details (Note that this lab uses "coordinator" instead of the paper's "master").

There are four files added for this part: `mr_protocol.h`, `mr_sequential.cc`, `mr_coordinator.cc`, `mr_worker.cc`.

Task 1

- `mr_sequential.cc` is a sequential mapreduce implementation, running Map and Reduce once at a time within a single process.
- You task is to implement the Mapper and Reducer for Word Count in `mr_sequential.cc`.

Task 2

- Your task is to implement a distributed MapReduce, consisting of two programs, `mr_coordinator.cc` and `mr_worker.cc`. There will be only one coordinator process, but one or more worker processes executing concurrently.
- The workers should talk to the coordinator via the `RPC`. One way to get started is to think about the RPC protocol in `mr_protocol.h` first.
- Each worker process will ask the coordinator for a task, read the task's input from one or more files, execute the task, and write the task's output to one or more files.
- The coordinator should notice if a worker hasn't completed its task in a reasonable amount of time, and give the same task to a different worker.
- In a real system, the workers would run on a bunch of different machines, but for this lab you'll run them all on a single machine.
- MapReduce relies on the workers sharing a file system. This is why we ask you to implement a global distributed ChFS in the first place.

Hints

- The number of Mappers equals to the number of files be to processed. Each mapper only processes one file at one time.
- The number of Reducers equals is a fixed number defined in `mr_protocol.h`.
- The basic loop of one worker is the following: ask one task (Map or Reduce) from the coordinator, do the task and write the intermediate key-value into a file, then submit the task to the coordinator in order to hint a completion.
- The basic loop of the coordinator is the following: assign the Map tasks first; when all Map tasks are done, then assign the Reduce tasks; when all Reduce tasks are done, the `Done()` loop returns true indicating that all tasks are completely finished.
- Workers sometimes need to wait, e.g. reduces can't start until the last map has finished. One possibility is for workers to periodically ask the coordinator for work, sleeping between each request. Another possibility is for the relevant RPC handler in the coordinator to have a loop that waits.
- The coordinator, as an RPC server, should be concurrent; hence please don't forget to lock the shared data.
- The Map part of your workers can use a hash function to distribute the intermediate key-values to different files intended for different Reduce tasks.
- A reasonable naming convention for intermediate files is mr-X-Y, where X is the Map task number, and Y is the reduce task number. The worker's map task code will need a way to store intermediate key/value pairs in files in a way that can be correctly read back during reduce tasks.
- Intermediate files will be operated on the Linux file system (part A, part B-a) or your distributed file system implemented in lab 1-3 (part B-b). If the file system's performance or your MapReduce implementation's performance is bad, it *shall not pass the test* !

Grading

Before grading, you need to check your lab3 implementation using the grade script in lab3. Your file system implemented in lab3 should work fine, or you shall not pass part B-b.

After you have implemented part1 & part2, run the grading script:

```
$ ./grade.sh
# ...
Passed part A (Word Count)
Passed part B-a (Word Count with distributed MapReduce)
Passed part B-b (Word Count with distributed MapReduce with performance requirements)
Lab4 passed

Passed all tests!
Score: 100/100
```

We will test your MapReduce following the evaluation criteria above.

Handin Procedure

After all above done:

```
% make handin
```

That should produce a file called lab4.tgz in the directory. Change the file name to your student id:

```
% mv lab4.tgz lab4_[your student id].tgz
```

Then upload **lab4_[your student id].tgz** file to Canvas before the deadline.

You'll receive full credits if your code passes the same tests that we gave you, when we run your code on our machines.