CS612 - Computer Systems

Distributed Systems - Assignment 1

Due: 26th September, 2021

In this assignment, we'll build a MapReduce system by implementing a worker process that calls application Map and Reduce functions and handles reading and writing files, and a master process that hands out tasks to workers and copes with failed workers. You need to setup Go (https://golang.org) to finish the assignment.

A simple sequential mapreduce implementation is provided at `src/mrsequential.go`. It runs the maps and reduces one at a time, in a single process. We also provide you with a couple of MapReduce applications: word-count in `src/wc.go`, and a text indexer in `src/indexer.go`. You can run word count sequentially as follows:

```
$ cd src
$ go build -race -buildmode=plugin wc.go
$ rm mr-out*
$ go run -race mrsequential.go wc.so pg*.txt
$ more mr-out-0
```

`mrsequential.go` leaves its output in the file `mr-out-0`. The input is from the text files named `pg-xxx.txt`.

The task is to implement a distributed MapReduce, consisting of two programs, the coordinator and the worker. There will be just one coordinator process, and one or more worker processes executing in parallel. The workers will talk to the coordinator via RPC. 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 (for this assignment, use ten seconds), and give the same task to a different worker. The "main" routines for the coordinator and worker are in `src/mrcoordinator.go` and `src/mrworker.go`; don't change these files. You should put your implementation in `src/mr/coordinator.go`, `src/mr/worker.go`, and `src/mr/rpc.go`. You can borrow code from `mrsequential.go`. You should also have a look at `wc.go` to see what MapReduce application code looks like.

Here's how to run your code on the word-count MapReduce application. First, make sure the word-count plugin is freshly built:

```
$ go build -race -buildmode=plugin wc.go
```

In the `src` directory, run the coordinator.

```
$ rm mr-out*
$ go run -race mrcoordinator.go pg-*.txt
```

The `pg-*.txt` arguments to `mrcoordinator.go` are the input files; each file corresponds to one "split", and is the input to one Map task. The `-race` flags runs go with its race detector. In one or more other windows, run some workers:

```
$ go run -race mrworker.go wc.so
```

When the workers and coordinator have finished, look at the output in `mr-out-*`. When you've completed the lab, the sorted union of the output files should match the sequential output, like this:

```
$ cat mr-out-* | sort | more
A 509
ABOUT 2
ACT 8
...
```

A test script is provided in `src/test-mr.sh`. The tests check that the `wc` and `indexer` MapReduce applications produce the correct output when given the `pg-xxx.txt` files as input. The tests also check that your implementation runs the Map and Reduce tasks in parallel, and that your implementation recovers from workers that crash while running tasks.

If you run the test script now, it will hang because the coordinator never finishes:

```
$ bash test-mr.sh
*** Starting wc test.
```

You can change `ret := false` to `true` in the `Done` function in `mr/coordinator.go` so that the coordinator exits immediately. Then:

```
$ bash test-mr.sh
*** Starting wc test.
sort: No such file or directory
cmp: EOF on mr-wc-all
--- wc output is not the same as mr-correct-wc.txt
--- wc test: FAIL
$
```

The test script expects to see output in files named `mr-out-X`, one for each reduce task. The empty implementations of `mr/coordinator.go` and `mr/worker.go` don't produce those files (or do much of anything else), so the test fails.

When you've finished, the test script output should look like this:

```
$ bash test-mr.sh
*** Starting wc test.
```

```
--- wc test: PASS
*** Starting indexer test.
--- indexer test: PASS
*** Starting map parallelism test.
--- map parallelism test: PASS
*** Starting reduce parallelism test.
--- reduce parallelism test: PASS
*** Starting crash test.
--- crash test: PASS
*** PASSED ALL TESTS
$
```

Things to Remember:

- You may have to set GOPATH and turn off GO111MODULE, depending on the machine you are working on, before getting the files to compile.
- The map phase should divide the intermediate keys into buckets for `nReduce` reduce tasks, where `nReduce` is the argument that `mrcoordinator.go` passes to `MakeCoordinator()`.
- The worker implementation should put the output of the n'th reduce task in the file `mr-out-n`. A `mr-out-n` file should contain one line per Reduce function output. The line should be generated with the Go `"%v %v"` format, called with the key and value. Have a look in `mrsequential.go` for the line commented "this is the correct format". The test script will fail if your implementation deviates too much from this format.
- You can modify `mr/worker.go`, `mr/coordinator.go`, and `mr/rpc.go`. You can temporarily modify other files for testing, but make sure your code works with the original versions.
- The worker should put intermediate Map output in files in the current directory, where your worker can later read them as input to Reduce tasks.
- `mrcoordinator.go` expects `mr/coordinator.go` to implement a `Done()` method that returns `true` when the MapReduce job is completely finished; at that point, `mrcoordinator.go` will exit.
- When the job is completely finished, the worker processes should exit. A simple way to implement this is to use the return value from call(): if the worker fails to contact the coordinator, it can assume that the coordinator has exited because the job is done, and so the worker can terminate too. Depending on your design, you might also find it helpful to have a "please exit" pseudo-task that the coordinator can give to workers.