6.824 - Spring 2020

6.824 Lab 1: MapReduce

Due: Feb 14 23:59

Introduction

In this lab you'll build a MapReduce system. You'll implement 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'll be building something similar to the MapReduce paper.

Collaboration Policy

You must write all the code you hand in for 6.824, except for code that we give you as part of assignments. You are not allowed to look at anyone else's solution, and you are not allowed to look at solutions from previous years. You may discuss the assignments with other students, but you may not look at or copy each others' code. The reason for this rule is that we believe you will learn the most by designing and implementing your lab solution yourself.

Please do not publish your code or make it available to current or future 6.824 students. `github.com` repositories are public by default, so please don't put your code there unless you make the repository private. You may find it convenient to use MIT's GitHub, but be sure to create a private repository.

Software

You'll implement this lab (and all the labs) in Go. The Go web site contains lots of tutorial information. We will grade your labs using Go version 1.13; you should use 1.13 too. You can check your Go version by running `go version`.

We recommend that you work on the labs on your own machine, so you can use the tools, text editors, etc. that you are already familiar with. Alternatively, you can work on the labs on Athena.

macOS

You can use Homebrew to install Go. After installing Homebrew, run `brew install go`.

Linux

Depending on your Linux distribution, you might be able to get an up-to-date version of Go from the package repository, e.g. by running `apt install golang`. Otherwise, you can manually install a binary from Go's website. First, make sure that you're running a 64-bit kernel (`uname -a` should mention "x86_64 GNU/Linux"), and then run:

```
$ wget -qO- https://dl.google.com/go/go1.13.6.linux-amd64.tar.gz | sudo tar xz -C /us
```

You'll need to make sure `/usr/local/bin` is on your `PATH`.

Windows

The labs probably won't work directly on Windows. If you're feeling adventurous, you can try to get them running inside Windows Subsystem for Linux and following the Linux instructions above. Otherwise, you can fall back to Athena.

Athena

You can log into a public Athena host with `ssh {your kerberos}@athena.dialup.mit.edu`. Once you're logged in, to get Go 1.13, run:

```
$ setup ggo
```

Getting started

You'll fetch the initial lab software with git (a version control system). To learn more about git, look at the Pro Git book or the git user's manual. To fetch the 6.824 lab software:

```
$ git clone git://g.csail.mit.edu/6.824-golabs-2020 6.824
$ cd 6.824
$ ls
Makefile src
$
```

We supply you with a simple sequential mapreduce implementation in `src/main/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 `mrapps/wc.go`, and a text indexer in `mrapps/indexer.go`. You can run word count sequentially as follows:

```
$ cd ~/6.824
$ cd src/main
$ go build -buildmode=plugin ../mrapps/wc.go
$ rm mr-out*
$ go run mrsequential.go wc.so pg*.txt
$ more mr-out-0
A 509
ABOUT 2
ACT 8
...
```

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

Feel free to borrow code from `mrsequential.go`. You should also have a look at `mrapps/wc.go` to see what MapReduce application code looks like.

Your Job

Your job is to implement a distributed MapReduce, consisting of two programs, the master and the worker. There will be just one master process, and one or more worker processes executing in parallel. 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. The workers will talk to the master via RPC. Each worker process will ask the master 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 master should notice if a worker hasn't completed its task in a reasonable amount of time (for this lab, use ten seconds), and give the same task to a different worker.

We have given you a little code to start you off. The "main" routines for the master and worker are in `main/mrmaster.go` and `main/mrworker.go`; don't change these files. You should put your implementation in `mr/master.go`, `mr/worker.go`, and `mr/rpc.go`.

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 -buildmode=plugin ../mrapps/wc.go
```

In the `main` directory, run the master.

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

The `pg-*.txt` arguments to `mrmaster.go` are the input files; each file corresponds to one "split", and is the input to one Map task.

In one or more other windows, run some workers:

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

When the workers and master 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
...
```

We supply you with a test script in `main/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 master never finishes:

```
$ cd ~/6.824/src/main
$ sh test-mr.sh
*** Starting wc test.
```

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

```
$ sh ./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/master.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:

```
$ sh ./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
$
```

You'll also see some errors from the Go RPC package that look like

```
2019/12/16 13:27:09 rpc.Register: method "Done" has 1 input parameters; needs exactly
```

Ignore these messages.

A few rules:

- The map phase should divide the intermediate keys into buckets for `nReduce` reduce tasks, where `nReduce` is the argument that `main/mrmaster.go` passes to `MakeMaster()`.
- The worker implementation should put the output of the X'th reduce task in the file `mr-out-X`.
- A `mr-out-X` 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 `main/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/master.go`, and `mr/rpc.go`. You can temporarily modify other files for testing, but make sure your code works with the original versions; we'll test 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.
- `main/mrmaster.go` expects `mr/master.go` to implement a `Done()` method that returns true when the MapReduce job is completely finished; at that point, `mrmaster.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 master, it can assume that the master 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 master can give to workers.

Hints

- One way to get started is to modify `mr/worker.go`'s `Worker()` to send an RPC to the master asking for a task. Then modify the master to respond with the file name of an as-yet-unstarted map task. Then modify the worker to read that file and call the application Map function, as in `mrsequential.go`.
- The application Map and Reduce functions are loaded at run-time using the Go plugin package, from files whose names end in `.so`.
- If you change anything in the `mr/` directory, you will probably have to re-build any MapReduce plugins you use, with something like `go build -buildmode=plugin ../mrapps/wc.go`

- This lab relies on the workers sharing a file system. That's straightforward when all workers run on the same machine, but would require a global filesystem like GFS if the workers ran on different machines.
- 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. One possibility is to use Go's `encoding/json` package. To write key/value pairs to a JSON file:

```
enc := json.NewEncoder(file)
for _, kv := ... {
  err := enc.Encode(&kv)
```

  and to read such a file back:

```
dec := json.NewDecoder(file)
for {
  var kv KeyValue
  if err := dec.Decode(&kv); err != nil {
    break
  }
  kva = append(kva, kv)
}
```

- The map part of your worker can use the `ihash(key)` function (in `worker.go`) to pick the reduce task for a given key.
- You can steal some code from `mrsequential.go` for reading Map input files, for sorting intermedate key/value pairs between the Map and Reduce, and for storing Reduce output in files.
- The master, as an RPC server, will be concurrent; don't forget to lock shared data.
- Use Go's race detector, with `go build -race` and `go run -race`. `test-mr.sh` has a comment that shows you how to enable the race detector for the tests.
- Workers will 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 master for work, sleeping with `time.Sleep()` between each request. Another possibility is for the relevant RPC handler in the master to have a loop that waits, either with `time.Sleep()` or `sync.Cond`. Go runs the handler for each RPC in its own thread, so the fact that one handler is waiting won't prevent the master from processing other RPCs.
- The master can't reliably distinguish between crashed workers, workers that are alive but have stalled for some reason, and workers that are executing but too slowly to be useful. The best you can do is have the master wait for some amount of time, and then give up and re-issue the task to a different worker. For this lab, have the master wait for ten seconds; after that the master should assume the worker has died (of course, it might not have).
- To test crash recovery, you can use the `mrapps/crash.go` application plugin. It randomly exits in the Map and Reduce functions.
- To ensure that nobody observes partially written files in the presence of crashes, the MapReduce paper mentions the trick of using a temporary file and atomically renaming it once it is completely written. You can use `ioutil.TempFile` to create a temporary file and `os.Rename` to atomically rename it.
- `test-mr.sh` runs all the processes in the sub-directory `mr-tmp`, so if something goes wrong and you want to look at intermediate or output files, look there.

Handin procedure

Important:

Before submitting, please run `test-mr.sh` one final time.

Use the `make lab1` command to package your lab assignment and upload it to the class's submission website, located at https://6824.scripts.mit.edu/2020/handin.py/.

You may use your MIT Certificate or request an API key via email to log in for the first time. Your API key (`XXX`) is displayed once you logged in, which can be used to upload lab1 from the console as follows.

```
$ cd ~/6.824
$ echo XXX > api.key
$ make lab1
```

Important:

Check the submission website to make sure it thinks you submitted this lab!

Note: You may submit multiple times. We will use the timestamp of your **last** submission for the purpose of calculating late days.

Please post questions on Piazza.