Lab Assignment 2: MapReduce

Notes

- This lab is assigned to students at MIT.
- You should work on this assignment individually.
- The deadline for parts I and II is March 8, 2017, and the deadline for parts III and IV is March 15, 2017.

Introduction

In this lab you'll build a MapReduce library as an introduction to programming in Go and to building fault tolerant distributed systems. In the first part you will write a simple MapReduce program. In the second part you will write a Master that hands out tasks to MapReduce workers, and handles failures of workers. The interface to the library and the approach to fault tolerance is similar to the one described in the original MapReduce paper.

Software

You'll implement this lab (and all the next labs) in Go. The Go web site contains lots of tutorial information which you may want to look at. It's probably most convenient for you to install Go on your own computer and do your development there. We will grade your labs using Go version 1.8.
We supply you with parts of a MapReduce implementation that supports both distributed and non-distributed operation (just the boring bits). 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, or, if you are already familiar with other version control systems, you may find this CS-oriented overview of git useful.
The URL for the course git repository is here. To install the files in your directory, you need to clone the course repository, by running the commands below.

```
$ git clone https://ielghand@bitbucket.org/ielghand/golabs-s2017.git golabs-S2017
$ cd golabs-S2017
$ ls
Makefile src
$
```

Git allows you to keep track of the changes you make to the code. For example, if you want to checkpoint your progress, you can commit your changes by running:

```
$ git commit -am 'partial solution to lab 2'
```

The Map/Reduce implementation we give you has support for two modes of operation, *sequential* and *distributed*. In the former, the map and reduce tasks are executed one at a time: first, the first map task is executed to completion, then the second, then the third, etc. When all the map tasks have finished, the first reduce task is run, then the second, etc. This mode, while not very fast, is useful for debugging. The distributed mode runs many worker threads that first execute map tasks in parallel, and then reduce tasks. This is much faster, but also harder to implement and debug.

Preamble: Getting Familiar with the Source

The mapreduce package provides a simple Map/Reduce library. Applications should normally call Distributed() [located in master.go] to start a job, but may instead call Sequential() [also in master.go] to get a sequential execution for debugging.
The code executes a job as follows:

1. The application provides a number of input files, a map function, a reduce function, and the number of reduce tasks (`nReduce`).

2. A master is created with this knowledge. It starts an RPC server (see `master_rpc.go`), and waits for workers to register (using the RPC call `Register()` [defined in `master.go`]). As tasks become available (in steps 4 and 5), `schedule()` [`schedule.go`] decides how to assign those tasks to workers, and how to handle worker failures.

3. The master considers each input file to be one map task, and calls `doMap()` [`common_map.go`] at least once for each map task. It does so either directly (when using `Sequential()`) or by issuing the `DoTask` RPC to a worker [`worker.go`]. Each call to `doMap()` reads the appropriate file, calls the map function on that file's contents, and writes the resulting key/value pairs to `nReduce` intermediate files. `doMap()` hashes each key to pick the intermediate file and thus the reduce task that will process the key. There will be `nMap x nReduce` files after all map tasks are done. Each file name contains a prefix, the map task number, and the reduce task number. If there are two map tasks and three reduce tasks, the map tasks will create these six intermediate files:

   ```
   mrtmp.xxx-0-0
   mrtmp.xxx-0-1
   mrtmp.xxx-0-2
   mrtmp.xxx-1-0
   mrtmp.xxx-1-1
   mrtmp.xxx-1-2
   ```

   Each worker must be able to read files written by any other worker, as well as the input files. Real deployments use distributed storage systems such as GFS to allow this access even though workers run on different machines. In this lab you'll run all the workers on the same machine, and use the local file system.

4. The master next calls `doReduce()` [`common_reduce.go`] at least once for each reduce task. As with `doMap()`, it does so either directly or through a worker. The `doReduce()` for reduce task `r` collects the `r`'th intermediate file from each map task, and calls the reduce function for each key that appears in those files. The reduce tasks produce `nReduce` result files.

5. The master calls `mr.merge()` [`master_splitmerge.go`], which merges all the `nReduce` files produced by the previous step into a single output.

6. The master sends a Shutdown RPC to each of its workers, and then shuts down its own RPC server.

Note: Over the course of the following exercises, you will have to write/modify `doMap`, `doReduce`, and `schedule` yourself. These are located in `common_map.go`, `common_reduce.go`, and `schedule.go`, respectively. You will also have to write the map and reduce functions in `../main/wc.go`.

You should not need to modify any other files, but reading them might be useful in order to understand how the other methods fit into the overall architecture of the system.

Part I: MapReduce Input and Output

The MapReduce implementation you are given is missing some pieces. Before you can write your first MapReduce function pair, you will need to fix the sequential implementation. In particular, the code we give you is missing two crucial pieces: the function that divides up the output of a map task, and the function that gathers all the inputs for a reduce task. These tasks are carried out by the `doMap()` function in `common_map.go`, and the `doReduce()` function in `common_reduce.go`, respectively. The comments in those files should point you in the right direction.
To help you determine if you have correctly implemented `doMap()` and `doReduce()`, we have provided you with a Go test suite that checks the correctness of your implementation. These tests are implemented in the file `test_test.go`. To run the tests for the sequential implementation that you have now fixed, run:

```
$ cd golabs-S2017
$ export ”GOPATH=$PWD” # go needs $GOPATH to be set to the project’s working directory
$ cd ”$GOPATH/src/mapreduce” $ go test-run Sequential ok mapreduce 2.694s $
```

Task: You receive full credit for this part if your software passes the Sequential tests (as run by the command above) when we run your software on our machines.

If the output did not show *ok* next to the tests, your implementation has a bug in it. To give more verbose output, set `debugEnabled = true` in `common.go`, and add `-v` to the test command above. You will get much more output along the lines of:

```
$ env ”GOPATH=$PWD/../../” go test -v -run Sequential
=== RUN TestSequentialSingle
master: Starting Map/Reduce task test
Merge: read mrtmp.test-res-0
master: Map/Reduce task completed
— PASS: TestSequentialSingle (1.34s)
=== RUN TestSequentialMany
master: Starting Map/Reduce task test
Merge: read mrtmp.test-res-0
Merge: read mrtmp.test-res-1
Merge: read mrtmp.test-res-2
master: Map/Reduce task completed
— PASS: TestSequentialMany (1.33s)
PASS
ok mapreduce 2.672s
```

Part II: Single-Worker Word Count

Now you will implement word count a simple Map/Reduce example. Look in `main/wc.go`, you'll find empty `mapF()` and `reduceF()` functions. Your job is to insert code so that `wc.go` reports the number of occurrences of each word in its input. A word is any contiguous sequence of letters, as determined by `unicode.IsLetter`.
We will use for testing the same files that we downloaded from the Gutenberg project and used in lab1. Here's how to run `wc` with the input files:

```
$ cd golabs-S2017
$ export "GOPATH=$PWD" $ cd "$GOPATH/src/main" $ go run wc.go master sequential pg-*.txt
# command-line-arguments ./wc.go:14: missing return at end of function ./wc.go:21: missing return at end of function
```

In the above screen-shot, the compilation fails because `mapF()` and `reduceF()` are not complete. Review Section 2 of the MapReduce paper. Your `mapF()` and `reduceF()` functions will differ a bit from those in the paper's Section 2.1. Your `mapF()` will be passed the name of a file, as well as that file's contents; it should split the contents into words, and return a Go slice of `mapreduce.KeyValue`. While you can choose what to put in the keys and values for the `mapF` output, for word count it only makes sense to use words as the keys. Your `reduceF()` will be called once for each key, with a slice of all the values generated by `mapF()` for that key. It must return a string containing the total number of occurrences of the key.

Hints:

- A good read on what strings are in Go is the Go Blog on strings.
- You can use `strings.FieldsFunc` to split a string into components.
- The strconv package (`http://golang.org/pkg/strconv/`) is handy to convert strings to integers etc.

You can test your solution using:

```
$ cd "$GOPATH/src/main"
$ time go run wc.go master sequential pg-*.txt
master: Starting Map/Reduce task wcseq
Merge: read mrtmp.wcseq-res-0
Merge: read mrtmp.wcseq-res-1
Merge: read mrtmp.wcseq-res-2
master: Map/Reduce task completed
14.59user 3.78system 0:14.81elapsed
```

The output will be in the file "mrtmp.wcseq". Your implementation is correct if the following command produces the output shown here:

```
$ sort -n -k2 mrtmp.wcseq | tail -10
he: 34077
was: 37044
that: 37495
I: 44502
in: 46092
a: 60558
to: 74357
of: 79727
and: 93990
the: 154024
```

You can remove the output file and all intermediate files with:

```
$ rm mrtmp.*
```

To make testing easy for you, run:

```
$ bash ./test-wc.sh
```

and it will report if your solution is correct or not.

Task: You receive full credit for this part if your MapReduce word count output matches the correct output for the sequential execution above when we run your software on our machines.

Part III: Distributing MapReduce Tasks

Your current implementation runs the map and reduce tasks one at a time. One of MapReduce's biggest selling points is that it can automatically parallelize ordinary sequential code without any extra work by the developer. In this part of the lab, you will complete a version of MapReduce that splits the work over a set of worker threads that run in parallel on multiple cores. While not distributed across multiple machines as in real MapReduce deployments, your implementation will use RPC to simulate distributed computation.

The code in `mapreduce/master.go` does most of the work of managing a MapReduce job. We also supply you with the complete code for a worker thread, in `mapreduce/worker.go`, as well as some code to deal with RPC in `mapreduce/common_rpc.go`.

Your job is to implement `schedule()` in `mapreduce/schedule.go`. The master calls `schedule()` twice during a MapReduce job, once for the Map phase, and once for the Reduce phase. `schedule()`'s job is to hand out tasks to the available workers. There will usually be more tasks than worker threads, so `schedule()` must give each worker a sequence of tasks, one at a time. `schedule()` should wait until all tasks have completed, and then return.

`schedule()` learns about the set of workers by reading its `registerChan` argument. That channel yields a string for each worker, containing the worker's RPC address. Some workers may exist before `schedule()` is called, and some may start while `schedule()` is running; all will appear on `registerChan`. `schedule()` should use all the workers, including ones that appear after it starts.

`schedule()` tells a worker to execute a task by sending a `Worker.DoTask` RPC to the worker. This RPC's arguments are defined by `DoTaskArgs` in `mapreduce/common_rpc.go`. The `File` element is only used by Map tasks, and is the name of the file to read; `schedule()` can find these file names in `mapFiles`.

Use the `call()` function in `mapreduce/common_rpc.go` to send an RPC to a worker. The first argument is the the worker's address, as read from `registerChan`. The second argument should be `"Worker.DoTask"`. The third argument should be the `DoTaskArgs` structure, and the last argument should be `nil`.

Your solution to Part III should only involve modifications to `schedule.go`. If you modify other files as part of debugging, please restore their original contents and then test before submitting.

To test your solution, you should use the same Go test suite as you did in Part I, but replace `-run Sequential` with `-run TestBasic`. This will execute the distributed test case without worker failures instead of the sequential ones you ran before:

```
$ go test -run TestBasic
```

Task: You receive full credit for this part if your software passes `TestBasic` from `test_test.go` (the test you run with the command above) when we run your software on our machines.

Hints:

- The documentation for the Go RPC package: `https://golang.org/pkg/net/rpc/`.
- `schedule()` should send RPCs to the workers in parallel so that the workers can work on tasks concurrently. You will find the go statement useful for this purpose; see Concurrency in Go.
- `schedule()` must wait for a worker to finish before it can give it another task. You may find Go's channels useful.
- You may find `sync.WaitGroup` useful.
- The easiest way to track down bugs is to insert print state statements (perhaps calling `debug()` in `common.go`), collect the output in a file with `go test -run TestBasic > out`, and then think about whether the output matches your understanding of how your code should behave. The last step (thinking) is the most important.
- To check if your code has race conditions, run Go's race detector with your test: `go test -race -run TestBasic > out`.

Note: The code we give you runs the workers as threads within a single UNIX process, and can exploit multiple cores on a single machine. Some modifications would be needed in order to run the workers on multiple machines communicating over a network. The RPCs would have to use TCP rather than UNIX-domain sockets; there would need to be a way to start worker processes on all the machines; and all the machines would have to share storage through some kind of network file system.

Handling Worker Failures

In this part you will make the master handle failed workers. MapReduce makes this relatively easy because workers don't have persistent state. If a worker fails, any RPCs that the master issued to that worker will fail (e.g., due to a timeout). Thus, if the master's RPC to the worker fails, the master

should re-assign the task given to the failed worker to another worker.
An RPC failure doesn't necessarily mean that the worker didn't execute the task; the worker may have executed it but the reply was lost, or the worker may still be executing but the master's RPC timed out. Thus, it may happen that two workers receive the same task, compute it, and generate output. Two invocations of a map or reduce function are required to generate the same output for a given input (i.e. the map and reduce functions are "functional"), so there won't be inconsistencies if subsequent processing sometimes reads one output and sometimes the other. In addition, the MapReduce framework ensures that map and reduce function output appears atomically: the output file will either not exist, or will contain the entire output of a single execution of the map or reduce function (the lab code doesn't actually implement this, but instead only fails workers at the end of a task, so there aren't concurrent executions of a task).

Note: You don't have to handle failures of the master. Making the master fault-tolerant is more difficult because it keeps state that would have to be recovered in order to resume operations after a master failure. Much of the later labs are devoted to this challenge.

Your implementation must pass the two remaining test cases in `test_test.go`. The first case tests the failure of one worker, while the second test case tests handling of many failures of workers. Periodically, the test cases start new workers that the master can use to make forward progress, but these workers fail after handling a few tasks. To run these tests:

```
$ go test -run Failure
```

Task: You receive full credit for this part if your software passes the tests with worker failures (those run by the command above) when we run your software on our machines.

Your solution to Part IV should only involve modifications to `schedule.go`. If you modify other files as part of debugging, please restore their original contents and then test before submitting.

Running All Tests

You can run all the tests by running the script `src/main/test-mr.sh`. With a correct solution, your output should resemble:

```
$ bash ./test-mr.sh
==> Part I
ok mapreduce 3.053s

==> Part I
I Passed test

==> Part III
ok mapreduce 1.851s

==> Part IV
ok mapreduce 10.650s
```

Collaboration Policy

In addition to the logistics we discussed at the first lecture, note the following:

- You must write all the code you hand in for this lab by yourself, except for code given to you as part of the assignment.
- 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 code yourself.
- Please do not publish your code or make it available on any public repository, such as `github.com`.
- Do not share your code with current or future students.

Deliverables

- Create a zip file from the golabs-S2017 directory. Rename the file to lab2_XX.zip, where XX is your ID.
- You need to send your code to CS432S17@gmail.com by the deadline at 8 AM. The subject line should be: "Lab Assignment 2 - SID:XX".
- The assignment will be discussed in the lab.