Project 1: MapReduce

Due: January 24th, 2022

Introduction

In this project, you will build a MapReduce library as a way to learn the Go programming language and as a way to learn about fault tolerance in distributed systems. In the first part, you will write a simple MapReduce program. In the second and third parts, you will write a Master that 1) hands out jobs to workers, and 2) handles failures of workers and of the network. The interface to the library and the approach to fault tolerance is similar to the one described in the original MapReduce paper.

Getting started

We supply you with a non-distributed MapReduce implementation, and a partial implementation of a distributed implementation (just the boring bits). Download the archive proj1.tar.gz and uncompress it. You will find an input file `kjv12.txt` in proj1/main, which was downloaded from here. Run our sequential implementation of MapReduce on this input file as follows:

```
$ cd proj1/main
$ go run main.go master kjv12.txt sequential
# mapreduce
../mapreduce/wc_impl.go:7: missing return at end of function
../mapreduce/wc_impl.go:12: missing return at end of function
```

The compiler produces two errors, because the implementation of the `WCMap` and `WCReduce` functions in `proj1/mapreduce/wc_impl.go` is incomplete.

Part I: Word count

Modify `WCMap` and `WCReduce` so that running the command listed above will compute the number of occurrences of each word in `kjv12.txt` in alphabetical order.

```
$ go run main.go master kjv12.txt sequential
Split kjv12.txt
DoMap: read split mrtmp.kjv12.txt-0 966967
DoMap: read split mrtmp.kjv12.txt-1 966941
DoMap: read split mrtmp.kjv12.txt-2 966974
DoMap: read split mrtmp.kjv12.txt-3 966970
DoMap: read split mrtmp.kjv12.txt-4 966905
DoReduce: read mrtmp.kjv12.txt-0-0
DoReduce: read mrtmp.kjv12.txt-1-0
DoReduce: read mrtmp.kjv12.txt-2-0
DoReduce: read mrtmp.kjv12.txt-3-0
DoReduce: read mrtmp.kjv12.txt-4-0
DoReduce: read mrtmp.kjv12.txt-0-1
DoReduce: read mrtmp.kjv12.txt-1-1
DoReduce: read mrtmp.kjv12.txt-2-1
DoReduce: read mrtmp.kjv12.txt-3-1
DoReduce: read mrtmp.kjv12.txt-4-1
DoReduce: read mrtmp.kjv12.txt-0-2
DoReduce: read mrtmp.kjv12.txt-1-2
DoReduce: read mrtmp.kjv12.txt-2-2
DoReduce: read mrtmp.kjv12.txt-3-2
DoReduce: read mrtmp.kjv12.txt-4-2
Merge: read mrtmp.kjv12.txt-res-0
Merge: read mrtmp.kjv12.txt-res-1
Merge: read mrtmp.kjv12.txt-res-2
```

The output will be in the file "output/mrtmp.kjv12.txt". Your implementation is correct if the top 1005 words match with those in "mr-testout.txt".

To make testing easy for you, run:

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

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

Your `WCMap()` and `WCReduce()` functions will differ a bit from those discussed in lecture and in the original MapReduce paper (Section 2.1). Your `WCMap()` will be passed some of the text from the file; it should split it into words and return a slice of key/value pairs, of type `mapreduce.KeyValue`. Your `WCReduce()` will be called once for each key, with a slice containing all the values generated by `WCMap()` for that key; it should return a single output value.

It will help to read our code for MapReduce, which is in `mapreduce.go` in package `mapreduce`. Look at `RunSingle()` and the functions it calls. This will help you to understand what MapReduce does and to learn Go by example.

Once you understand this code, implement `WCMap` and `WCReduce` in `mapreduce/wc_impl.go`.

You can use `strings.FieldsFunc` to split a string into components. You can consider a word to be any contiguous sequence of letters, as determined by `unicode.IsLetter`. A good read on what strings are in Go is the Go blog about strings. The strconv package (http://golang.org/pkg/strconv/) is handy to convert strings to integers etc.

Note: You can remove the output file and all intermediate files by clearing out the contents of the "output" subdirectory.

Part II: Distributing MapReduce jobs

In this part, you will complete a version of MapReduce that splits the work up over a set of worker threads, in order to exploit multiple cores. A master thread hands out work to the workers and waits for them to finish. The master should communicate with the workers via RPC (look at the invocation of `Worker.Shutdown` via RPC in `mapreduce/master.go` as an example). We give you the worker code (`mapreduce/worker.go`), the code that starts the workers (`mapreduce/mapreduce.go`), and code to deal with RPC messages (`mapreduce/rpcs.go`).

Your job is to complete `master_impl.go` in the `mapreduce` package. In particular, you should complete `RunMasterImpl()` in `master_impl.go` (which is invoked by `RunMaster()` in `master.go`) to hand out the Map and Reduce tasks to workers, and return only when all the tasks have finished.

Look at `Run()` in `mapreduce.go`. It calls `Split()` to split the input into per-map-job files, then calls `RunMaster()` to run the map and reduce jobs, then calls `Merge()` to assemble the per-reduce-job outputs into a single output file. `RunMasterImpl` only needs to tell the workers the name of the original input file (`mr.file`) and the job number; each worker knows from which files to read its input and to which files to write its output.

Each worker sends a Register RPC to the master when it starts. `mapreduce.go` already implements the master's `MapReduce.Register` RPC handler for you, and passes the new worker's information to `mr.registerChannel`. Your `RunMasterImpl` should process new worker registrations by reading from this channel.

Information about the MapReduce job is in the `MapReduce` struct, defined in `mapreduce.go`. To keep track of any additional state (e.g., the set of available workers), modify the `MapReduceImpl` and `WorkerInfoImpl` structs (defined in `mapreduce_impl.go` and `master_impl.go`) and initialize this additional state in the `InitMapReduceImpl()` function. The master does not need to know which Map or Reduce functions are being used for the job; the workers will take care of executing the right code for Map or Reduce.

You should run your code using Go's unit test system. We supply you with a set of tests in `mapreduce_test.go`. You run unit tests in a package directory (e.g., the mapreduce directory) as follows:

```
$ cd proj1/mapreduce
$ go test
```

Note: You can ignore the error messages such as the following that will show up at the beginning of your test output.

```
2021/01/10 13:20:22 method CleanupFiles has wrong number of ins: 1
2021/01/10 13:20:22 method CleanupRegistration has wrong number of ins: 1
2021/01/10 13:20:22 method FindIdleWorker has wrong number of ins: 1
2021/01/10 13:20:22 method InitMapReduceImpl has wrong number of ins: 5
2021/01/10 13:20:22 method KillWorkers has wrong number of ins: 1
2021/01/10 13:20:22 method Merge has wrong number of ins: 1
2021/01/10 13:20:22 method Run has wrong number of ins: 1
2021/01/10 13:20:22 method RunMaster has wrong number of ins: 1
2021/01/10 13:20:22 method RunMasterImpl has wrong number of ins: 1
2021/01/10 13:20:22 method RunPhase has wrong number of ins: 2
2021/01/10 13:20:22 method Split has wrong number of ins: 2
2021/01/10 13:20:22 method StartRegistrationServer has wrong number of ins: 1
```

You are done with Part II when your implementation passes the first two tests ("Basic mapreduce" and "Slow Worker mapreduce") in `mapreduce_test.go` in the `mapreduce` package. You don't yet have to worry about failures of workers.

The master should send RPCs to the workers in parallel so that the workers can work on jobs concurrently. You will find the `go` statement useful for this purpose and the Go RPC documentation.

The master may have to wait for a worker to finish before it can hand out more jobs. You may find channels useful to synchronize threads that are waiting for a reply. Channels are explained in the document on Concurrency in Go.

The easiest way to track down bugs is to insert log.Printf() statements, collect the output in a file with `go test > 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.**

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.

Part III: Handling worker failures and network unreliability

In this part, you will make the master cope with failed workers and an unreliable network. MapReduce makes this relatively easy because workers don't have persistent state. If a worker fails or if the network connectivity between the master and 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 a worker fails, the master should re-assign the job to another worker.

An RPC failure doesn't necessarily mean that the worker failed; the worker may just be unreachable but still computing. Thus, it may happen that two workers receive the same job and compute it. However, because jobs are idempotent, it doesn't matter if the same job is computed twice---both times it will generate the same output. So, you don't have to do anything special for this case. (Our tests never fail workers in the middle of a job, so you don't even have to worry about several workers writing to the same output file.)

You don't have to handle failures of the master; we will assume it won't fail. Making the master fault-tolerant is more difficult because it keeps persistent state that would have to be recovered in order to resume operations after a master failure. Much of the later projects are devoted to this challenge.

Your implementation must pass the remaining test cases in `mapreduce_test.go`, which test your handling of the failure of one worker, the failures of many workers, and different levels of network unreliability.

Handin procedure

Clone the uniqname.1 repository that we have created for you on GitHub. If you haven't already done so, declare your GitHub ID.

When you submit your project to the autograder, it will pull the following three files from your repository:

- `mapreduce/wc_impl.go`
- `mapreduce/mapreduce_impl.go`
- `mapreduce/master_impl.go`

So, please ensure that a) your repository has a directory `mapreduce` containing these three files, and b) all modifications that you make to the code handed out are restricted to only these three files.

You can check that the set of test cases that your submission passes on the autograder matches those that your code passes locally on your computer. If you find a discrepancy,

- Check that you have only modified the *impl.go files
- Check that you pass on CAEN all the tests that you pass locally
- Ensure that you use appropriate synchronization to protect any access to shared data from any goroutine; race conditions in your code can cause your code to behave differently on the autograder than on CAEN
- Since some of the tests are non-deterministic, run them repeatedly and make sure you pass them every time
- If you still pass all runs of the test cases, post privately on Piazza

You will receive full credit if your software passes the `mapreduce_test.go` tests when we run your software on our machines. We will use the timestamp of your **last** submission for the purpose of calculating late days.