Project 3: MapReduce

Introduction

In this project you'll 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 part you will write a Master that hands out jobs to 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.

Getting started

There is an input file kjv12.txt in ~/p3/src/main, which was downloaded from here. Compile the initial software we provide you and run it with the downloaded input file:

```
$ export GOPATH=$HOME/p3
$ cd ~/p3/src/main
$ go run wc.go master kjv12.txt sequential
# command-line-arguments
./wc.go:11: missing return at end of function
./wc.go:15: missing return at end of function
```

The compiler produces two errors, because the implementation of the Map and Reduce functions is incomplete.

Part I: Word count

Modify Map and Reduce so that wc.go reports the number of occurrences of each word in alphabetical order.

```
$ go run wc.go master kjv12.txt sequential
Split kjv12.txt
Split read 4834757
DoMap: read split mrtmp.kjv12.txt-0 966954
DoMap: read split mrtmp.kjv12.txt-1 966953
DoMap: read split mrtmp.kjv12.txt-2 966951
DoMap: read split mrtmp.kjv12.txt-3 966955
DoMap: read split mrtmp.kjv12.txt-4 966944
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 phaseMerge: 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 "mrtmp.kjv12.txt". Your implementation is correct if the following command produces the following top 10 words:

```
$ sort -n -k2 mrtmp.kjv12.txt | tail -10
unto: 8940
he: 9666
shall: 9760
in: 12334
that: 12577
And: 12846
to: 13384
of: 34434
and: 38850
the: 62075
```

To make testing easy for you, run:

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

and it will report if your solution is correct or not.
Before you start coding read Section 2 of the MapReduce paper. Your Map() and Reduce() functions will differ a bit from those in the paper's Section 2.1. Your Map() will be passed some of the text from the file; it should split it into words, and return a list.List of key/value pairs, of type mapreduce.KeyValue. Your Reduce() will be called once for each key, with a list of all the values generated by Map() 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 well help you to understand what MapReduce does and to learn Go by example.
Once you understand this code, implement Map and Reduce in wc.go.

Hint: you can use strings.FieldsFunc to split a string into components.
Hint: for the purposes of this exercise, 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 on strings.
Hint: the strconv package (http://golang.org/pkg/strconv/) is handy to convert strings to integers etc.

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

```
$ rm mrtmp.*
```

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. We give you the worker code (mapreduce/worker.go), the code that starts the workers, and code to deal with RPC messages (mapreduce/common.go).

Your job is to complete master.go in the mapreduce package. In particular, you should modify RunMaster() inmaster.go to hand out the map and reduce jobs to workers, and return only when all the jobs have finished.
Look at Run() in mapreduce.go. It calls Split() to split the input into per-map-job files, then calls your RunMaster() to run the map and reduce jobs, then calls Merge() to assemble the per-reduce-job outputs into a single output file.RunMaster 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'sMapReduce.Register RPC handler for you, and passes the new worker's information to mr.registerChannel. YourRunMaster should process new worker registrations by reading from this channel.
Information about the MapReduce job is in the MapReduce struct, defined in mapreduce.go. Modify the MapReduce struct to keep track of any additional state (e.g., the set of available workers), and initialize this additional state in theInitMapReduce() 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 test_test.go. You run unit tests in a package directory (e.g., the mapreduce directory) as follows:

```
$ cd mapreduce
$ go test
```

You are done with Part II when your implementation passes the first test (the "Basic mapreduce" test) intest_test.go in the mapreduce package. You don't yet have 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 reply with the master once the reply arrives. 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

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 job given to the failed worker 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 anything special for this case. (Our tests never fail workers in the middle of 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.

Your implementation must pass the two remaining test cases in test_test.go. The first case tests the failure of one worker. 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 jobs.