

CS134 - Assignment 1: MapReduce

Due: Friday, April 10, 10:00PM PT

Introduction

In this assignment (which will be completed **individually**), 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](#).

Collaboration Policy

You must write all the code you hand in for CS134, except for code that we give you as part of the assignment. You may discuss clarifications about the assignments with other students, but you may not discuss solutions or look at or copy each others' code. Please do not publish your code or make it available to future CS134 students -- for example, please do not make your code visible on GitHub.

Setting up the requirements

- Install Go: You'll implement this assignment (and all the other assignments) in [Go](#). The Go website contains lots of tutorial information which you may want to look at. You can find the installation instructions [here](#).
- Install Git: We supply you with a non-distributed MapReduce implementation, and a partial implementation of a distributed implementation (skeleton code with utility components). You'll fetch the initial implementations with [git](#) (a version control system). To learn more about git, look at 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.
- Setup a private Git repository: To start this assignment, you should duplicate the (public) assignment 1 git repository into a private repository of your own. The assignment 1 repository can be found here: [S20-CS134/assignment1-skeleton](#). Your own private copy of the assignment 1 repository is called a mirror repository. Each individual student should have their own private mirror repository where they commit their changes. When the assignment is complete, students will submit a link to their private mirror repository in the submission form (listed below).

1. First, you should create a new **private** repository on GitHub. Let's assume the name of this repository is *assignment1*. Also, replace `${USER_NAME}` with your GitHub user name while running the following commands.

2. Next, create a bare mirrored clone of *S20-CS134/assignment1-skeleton* repository.

```
$ git clone --mirror git@github.com:S20-CS134/assignment1-skeleton.git
```

3. Then, you should mirror-push to the new repository, in addition to setting the push location to your private repository:

```
$ cd assignment1-skeleton.git
$ git push --mirror git@github.com:${USER_NAME}/assignment1.git
$ git remote set-url --push origin git@github.com:${USER_NAME}/assignment1.git
```

4. To finish this step, remove the temporary local repository you created in the second step:

```
$ cd ..
$ git clone git@github.com:${USER_NAME}/assignment1.git
$ rm -rf assignment1-skeleton.git
$ cd assignment1/
$ git remote -v
origin  git@github.com:${USER_NAME}/assignment1.git (fetch)
origin  git@github.com:${USER_NAME}/assignment1.git (push)
```

Now you have your own copy of the original *S20-CS134/assignment1-skeleton* repository. Please add your designated TA as a collaborator on this private repository. Your designated TA is the one leading your discussion section; please email the TAs if you are unsure about whom to add.

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 and push your changes by running:

```
# make some changes
$ git status
```

```
$ git commit -am 'Added partial solution to assignment 1'
$ git push
```

- Note: Go and Git are already installed on the seasnet linux servers. We will be using lnxsrv09 or equivalent to grade these projects for consistency.

Getting started

There is an input file `kjv12.txt` in `src/main`, which was downloaded from [here](#). Assuming you are in the `assignment1` directory, compile the initial code we provide you and run it with the downloaded input file:

```
$ export GOPATH=$(pwd)
$ cd src/main
$ go run wc.go master kjv12.txt sequential
# command-line-arguments
./wc.go:15:1: missing return at end of function
./wc.go:21:1: missing return at end of function
```

The compiler produces two errors, because the implementation of the `Map()` and `Reduce()` functions are incomplete.

Part I: Word count

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

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 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 will 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`.

After you finish implementing the `Map()` and `Reduce()` functions, your command line output should be similar to the following:

```
$ 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 actual output of the word count program 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.

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()` in `master.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's `MapReduce.Register` RPC handler for you, and passes the new worker's information to `mr.registerChannel`. Your `RunMaster` 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 the `InitMapReduce()` 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
```

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.

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

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 do 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. Much of the later assignments are devoted to this challenge.

Your implementation must pass the two remaining test cases in `test_test.go`. The first test 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.

Assignment Submission

To submit the assignment, please push your final code into your repository. Then fill out the [submission form](#) to turn in your assignment. You may submit up to 5 times before the assignment is due. We will evaluate the code specified in your **last** submission prior to the 4/10 10pm PT deadline.

You will receive full credit if your code passes the `test_test.go` tests when we run your software on the SEASnet machines; we encourage you to develop or at least test your code on the SEASnet machines prior to submission.

Remember that late days cannot be used for this assignment.

Please post questions on [Piazza](#).