Lab 2: MapReduce. Can be done in groups of up to three students

In this lab you'll build a simplified MapReduce system based on the article covered in class (MapReduce paper). 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 implement this lab in Go.

Getting started

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 MapReduce application: word-count in `mrapps/wc.go`. You can run word count sequentially as follows:

```
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

...

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.
You have two options:

a. The workers communicate with the server via channels
b. The workers will talk to the master via RPC. A good tutorial in RPC using go can be found here: https://ipfs.io/ipfs/QmfYeDhGH9bZzihBUDEQbCbTc5k5FZKURMUoUvfmc27BwL/rpc/go_rpc.html

Each team select the option a or b , I have no preference , usually people that already took networking prefers b.

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.

Your application should more or less work like this (If it doesn't than create your own instructions to run and test):

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.
Hints:

- 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.

- One way to get started is to send a request 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`.
- 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)
  }
```

- 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, 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.
- Failure detector: 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).

  You can do this Fail Detection using a Gossip Protocol as the one discussed in class and get 20 points towards the Final exam, Requirements for Gossip Membership Heartbeat Protocol:

  Have 8 computing nodes each with two neighbors (can select them at random or keep a fix to whom they need to exchange heartbeat tables with. Heartbeat tables contain : id neighbor, hbcounter,time.

  Every node: Keep a Hbcounter gets increase every X amount of time (you choose). Send their HB tables to its neighbor every Y amount of time (you choose). Simulate one node failing every Z amount of time and how the tables change

- 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.

Upload to canvas a zip file with : code, readme document (Names of the group, how to run the code, screen shots of your program output for wc)