Project 3: The MapReduce Library

This is a group project. Known issue: the starter code might not work on OSX or Windows. You will have to use terminal to run the test cases inside an Ubuntu machine setup previously.

In this project, you will build a MapReduce Library as a way to gain basic understanding of distributed systems including RPC and fault tolerance. This also gives you another opportunity to practice the Go programming language.

The project is divided into four phases. In the first and second phases, you will implement a **sequential** MapReduce library and write an application called **Word Count** that uses the MapReduce library. The third and fourth phases, you will implement a **distributed** MapReduce library using Go RPC and goroutines that tolerates worker failures.

The interface to the library is similar to the one described in the original MapReduce paper. To help you get started, we provide you with a significant amount of scaffolding code with necessary inline comments. We also test the starter code, together with a reference implementation, in a Ubuntu 16.04 machine.

Overview

The **starter code** is structured as following:

- `src/mapreduce/`: this directory consists of all source codes for our simplified MapReduce Library.
- `src/main/`: this directory consists of the starter code for word count application (`wc.go`), input files `pg-*txt`, and bash scripts.
- `src/go.mod`: specifies the module name and go version
- `README.txt`.

For this project, you will be asked to add code to places that say **TODO:** across multiple go files. Before starting the implementation, we highly recommend you re-fresh your understanding about MapReduce by reading the original MapReduce paper, in conjunction with the starter code. Reading the provided MapReduce implementation might be useful to understand how the other methods fit into the overall architecture of the system hierarchy.

MapReduce Workflow Summary

Spoiler alert: you should skip this section if you want to understand the system architecture by reading the source code yourself.

We make a few simplifying assumptions for the MapReduce Library in order to focus on practicing important distributed system concepts without requiring error-prone manual setups. 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.

Below, we summarize the workflow of our simplified MapReduce Library.

1. The application, e.g., the word count program, provides a number of input files, a map function, a reduce function, and the number of reduce tasks (`nReduce`).
2. The master will be created with the above information. Further, it will start an RPC server (see `mapreduce/master_rpc.go`) which will wait for the workers to register. The worker registers with the master by calling the RPC `Register()`, defined in `mapreduce/master.go`. Further, workers will also start RPC servers so that they can wait for the master to dispatch them the tasks.
3. As tasks become available, the master leverages the `schedule()` (see `mapreduce/schedule.go`) to decide how to assign those tasks to workers, and how to handle worker failures.
4. The master considers each input file one map task, and makes a call to `doMap()` in `mapreduce/common_map.go` at least once for each task. Depending on the execution mode, i.e., **sequential** or **distributed**, the master either calls `doMap()` directly or via the `DoTask()` RPC defined in `mapreduce/worker.go` to dispatch the current task to the corresponding worker.
5. Each call to `doMap()` reads the appropriate file, calls the map function on that file's contents, and produces `nReduce` files for each map file. Specifically, for the ith map task, it will generate a list of files with the following naming pattern: `fi-0`, `fi-1` ... `fi-[nReduce-1]`. Thus, after all map tasks are done, the total number of files will be the product of the number of files and `nReduce`.
6. After all map tasks are completed, the master next makes a call to `doReduce()` in `mapreduce/common_reduce.go` at least once for each reduce task. Similar to `doMap()`, the master either calls `doReduce()` directly or via the RPC. `doReduce()` collects corresponding files generated at the map phase. Specifically, for the jth reduce tasks, it will run the reduce function on files with the following naming pattern: textttf0-j, `1-j` ... `f[n-1]-j`. After all reduce tasks finish, we will end up with `nReduce` result files.
7. The master calls `mr.merge()` in `mapreduce/master_splitmerge.go`, which merges all the `nReduce` files produced by the previous step into a single output.
8. Finally, the master sends a Shutdown RPC to each of its workers, and then shuts down its own RPC server.

Phase 1: A Sequential MapReduce Library

In this phase, you will implement a sequential version of the MapReduce Library. This is a useful step for understanding the MapReduce workflow, as well as for debugging—as it removes much of the noise seen in the distributed execution.

In essence, both the map and reduce tasks will be executed sequentially. If there are N map tasks, the master will run the first map task to completion, then the second, then the third, until all the N map tasks have finished. Afterward, the master will launch the first reduce task, then the second, until all the reduce tasks have finished.

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 `mapreduce/common_map.go`, and the `doReduce()` function in `mapreduce/common_reduce.go` respectively. The comments in those files should point you in the right direction.

For this Phase, you will **only** write/modify **1)** `doMap()` function in `mapreduce/common_map.go`, **2)** and the `doReduce()` function in `mapreduce/common_reduce.go`.

Testing

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, follow this (or a non-bash equivalent) sequence of shell commands, starting from the `starter-project-mr/` directory:

```
$ cd src/mapreduce
$ go test -v -run Sequential  mr/mapreduce
=== RUN   TestSequentialSingle
--- PASS: TestSequentialSingle (8.00s)
=== RUN   TestSequentialMany
--- PASS: TestSequentialMany (9.07s)
PASS
ok   mr/mapreduce  17.076s
```

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 `mapreduce/common.go`, and add -v to the test command above.

Phase 2: Word Count Application

Before you start implementing the word count, read Section 2 of the MapReduce paper to refresh your conceptual understanding. For this phase, You will **only** write/modify the `mapF()` and `reduceF()` functions in the `main/wc.go`. However, your `mapF()` and `reduceF()` functions will differ a bit from those in the paper's Section 2.1, for simplicity. Specifically, your `mapF()` will be passed the name of a file, as well as that file's contents; it should split it into words, and return a Go slice of key/value pairs, of type `mapreduce.KeyValue`. Your `reduceF()` will be called once for each key, with a slice of all the values generated by `mapF()` for that key; it should return a single output value.

Testing

To test the word count application, you can take the following three steps:

First, type the following commands into the terminal. If `debugEnabled` is set to `true` in `mapreduce/common.go`, a working implementation should produce similar outputs. The final word count result will be stored in a file called `mrtmp.wcseq`.

```
$ cd src/main
$ go run wc.go master sequential pg-*.txt
master: Starting Map/Reduce task wcseq
Merge phaseMerge: read mrtmp.wcseq-res-0
Merge: read mrtmp.wcseq-res-1
Merge: read mrtmp.wcseq-res-2
master: Map/Reduce task completed
```

Second, check if the top-10 most frequent words are correctly identified by typing this following command into the terminal.

```
$ 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
```

Third, compare the terminal output from the second step to the sample result found in `main/mr-testout.txt`.

Alternatively, as a one-step test, you can leverage the provided bash script `./test-wc.sh`. If you have set `debugEnabled` to be `true`, you should observe the following output where the last line indicates that you have **passed test case**.

```
$ cd src/main
$ ./test-wc.sh
master: Starting Map/Reduce task wcseq
```

```
Merge phaseMerge: read mrtmp.wcseq-res-0
Merge: read mrtmp.wcseq-res-1
Merge: read mrtmp.wcseq-res-2
master: Map/Reduce task completed
passed test case!
```

Phase 3: Distributed MapReduce

In this phase, you will write a distributed MapReduce library, so that a master can hands out tasks to multiple workers. By doing so, the MapReduce library will take advantage of the hardware resources (e.g., number of CPU cores) by first running the map tasks in parallel and then the reduce tasks, also in parallel. In essence, you will implement the Distributed MapReduce library so that developers can simply submit a job (e.g., the word count application) without worrying about running the job in parallel on many machines.

You will **only** have to modify `schedule()` from `schedule.go`. Specifically, you should modify `schedule()` to hand out the map and reduce tasks to workers, and return only when all the tasks have finished. As with the previous phases of this assignment, 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.

Note that for simplicity, the work is **NOT** distributed across multiple machines as in real-world Map/Reduce deployments; rather, your implementation will be using **RPC** and **channels** to simulate a distributed environment.

An important restriction is that the master should only communicate with the workers via RPC, as in many real-world distributed systems. As part of the starter code, we provide you with the worker code `mapreduce/worker.go`, the code to start the workers, and the code to deal with RPC messages (`mapreduce/common_rpc.go`).

Optional: if you are curious on how much speedup the distributed MapReduce provides over the Sequential counterpart, you can instrument the `master.go` to log the time. Where, you might ask? Well, that is for you to figure out. I am sure you can! We provide a bash script `src/main/test-wc-distributed.sh` that helps you launch both the master and the workers inside the Ubuntu 16.04 VM.

Hints

Spoiler alert: you should skip this section if you want to understand the system architecture by reading the source code yourself.

There are three logical roles in a canonical MapReduce Framework: the master, the mapper, and the reducer. The master is considered a central coordinator that orchestrates the job execution by dividing a job to many tasks, and assigning tasks to many workers. If a worker is currently executing a map function, it assumes the role of a mapper; similarly, a reducer is a worker that executes the reduce function.

At the high level, the master thread hands out work to the workers and waits for them to finish, as well as coordinates the parallel execution of tasks. Information about the currently running job is in the `Master` struct, defined in `master.go`. Note that 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 functions (the correct functions are given to them when they are started by `main/wc.go`).

When the master starts, it starts a RPC server which allows workers to register themselves with the master. When a worker starts, it calls the `Register` RPC (exposed by the master RPC server) to pass the new worker's information to the `registerChannel`. Your `schedule()` should process new worker registrations by reading from this channel. Again, the work distribution via `schedule()` should be in parallel—the worker `DoTask` should not block scheduling the next task.

The master tells the worker about a new task by using the RPC call `Worker.DoTask`, giving a `DoTaskArgs` object as the RPC argument. The worker, once receiving the `DoTaskArgs`, will assume either the mapper or

the reducer role based on the `DoTaskArgs.Phase`. Further, each worker knows from which files to read its input and to which files to write its output.

Testing

To test your distributed MapReduce implementation, you should use the same Go test suite as in previous phases but with the following command:

```
$ cd src/mapreduce
$ go test -run TestBasic mr/mapreduce
ok   mr/mapreduce  16.282s
```

Alternatively, if you need more verbose output, you should set `debugEnabled` in `mapreduce/common.go` to `true` and use the following command to save the output. You should see similar information as shown below.

```
$ go test -v -run TestBasic mr/mapreduce > testbasic.log
$less testbasic.log
=== RUN   TestBasic
2021/04/18 15:42:18 rpc.Register: method "CleanupFiles" has 1 input parameters; needs
 exactly three
2021/04/18 15:42:18 rpc.Register: method "Lock" has 1 input parameters; needs exactly
 three
2021/04/18 15:42:18 rpc.Register: method "Unlock" has 1 input parameters; needs exactly
 three
[...truncated...]
Merge: read mrtmp.test-res-48
Merge: read mrtmp.test-res-49
/var/tmp/824-1000/mr7582-master: Map/Reduce task completed
--- PASS: TestBasic (20.17s)
PASS
ok   mr/mapreduce
```

Phase 4: Handling worker failures

Machines can fail, especially in large-scale distributed systems. In this phase, you will modify the `schedule.go` to make the master handle failed workers. For simplicity, we assume that master will never 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.

In MapReduce, this is relatively easy as workers do not have persistent state. If a worker fails, any outstanding RPCs that the master issued to the worker will fail (e.g., due to a timeout). Therefore, 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 failed; the worker may just be unreachable but still computing. Thus, it may happen that two workers receive the same task and compute it. However, because tasks are idempotent, it doesn't matter if the same task 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 task, so you don't even have to worry about several workers writing to the same output file.)

Testing

To test your worker fault-tolerance implementation, you should use the following command:

```
$ cd src/
$ go test -run Failure mr/mapreduce
ok   mr/mapreduce  36.635s
```

There are two test cases associated with this phase: the first case tests the failure of one worker, while the second test case tests handling of many failures of workers—periodically, the test case starts new workers that the master can use to make forward progress, but these workers fail after handling a few tasks.

Checkpoint Contributions

Students must submit work that demonstrates substantial progress towards completing the project on the checkpoint date. Substantial progress is judged at the discretion of the grader to allow students flexibility in prioritizing their efforts. However, as an example, any checkpoint assignment which passes **Phase 1 and Phase 2** will receive full credit towards the checkpoint. **Projects that fail to submit a checkpoint demonstrating significant progress will incur a 10% penalty during final project grading.**

Deliverables and Grading

Each group should submit via the corresponding Canvas assignment page. To better support students' individual need, we allow students to switch groups for this project. Therefore, we kindly ask you to keep the group information up-to-date by claiming a group slot in the **project-mapreduce** on Canvas.

When submitting the project, please make sure to include the following:

- Go source code files that contains your **implementations** and reasonable **inline documentations** for all parts.
- `README.txt` that contains your brief answers to the prompts, and anything that you feel the grader should know when grading the project. Only plaintext write-ups are accepted; Markdown is allowed.
- All other provided starter files in **their original states and directories**. Changes to these starter files risk failing the test cases.

Please compress all the files together as a single .zip, named `YOUR_GROUP_ID_project_mapreduce.zip`, archive for submission. As with all projects, please **only use standard zip files** for compression; **.rar, .7z, and other custom file formats will not be accepted**.

Grading

A grading rubric has been provided at following table to give you a guide for how the project will be graded.

Task ID	Description	Points
1	Passed the `TestSequential*()` tests defined in `test_test.go`;	24
2	Passed the test for the Word Count application via `test-wc.sh`;	24
3	Passed `TestBasic()` test for the Distributed MapReduce	32
4	Passed the `TestOneFailure()` and `TestManyFailures()` tests	24
5	The design document (as part of README.txt)	12

Note: The final project grade will also be adjusted based on the checkpoint completion as well as the late submission.