Term Project. MapReduce with RPC in Go

SWE4004 Principles of Distributed Computing - Spring 2024

Due date: Jun 5 (Mon) 11:59pm

1 In-Ui-Ye-Ji Cluster

- The project codes are provided in /home/swe4004/gomr in In-Ui-Ye-Ji cluster. In-Ui-Ye-Ji cluster has four nodes. `swin.skku.edu`, `swui.skku.edu`, `swye.skku.edu`, and `swji.skku.edu`. These four nodes share the same NFS home directory and NIS account. To login from outside the university network, you need to change the SSH connection port to 1398. For example, `$ ssh swin.skku.edu -p 1398 -l s20333555`
- The cluster account ID is what you get by replacing the first two digits of the student ID with 's'. For example, if your student ID is 2020333555, the account ID is s20333555. The initial password is what you replace 's' with 'pw'. For example, pw20333555 is the initial password for 2020333555.

2 Sequential MapReduce

For this project, you will be provided with parts of a MapReduce implementation. It has support for two modes of operation, sequential and distributed. The first part deals with the former. The map and reduce tasks are all executed in serial: the first map task is executed to completion, then the second, then the third, etc. When all the map tasks have finished, the first reduce task is run, then the second, etc. This mode, while not very fast, can be very useful for debugging, since it removes much of the noise seen in a parallel execution. The sequential mode also simplifies or eliminates various corner cases of a distributed system.

The mapreduce package (located at `/home/swe4004/gomr/sequential_mapreduce/src/mapreduce`) provides a simple Map/Reduce library with a sequential implementation. In normal cases, applications would call `Distributed()` — located in `mapreduce/master.go` — to start a job, but may instead call `Sequential()` — also in `mapreduce/master.go` — to get a sequential execution.

The flow of the mapreduce implementation is as follows:

- The application provides a number of input files, a map function, a reduce function, and the number of reduce tasks (`nReduce`).
- A master is created with this knowledge. It spins up an RPC server (see `mapreduce/master_rpc.go`), and waits for workers to register (using the RPC call `Register()` defined in `mapreduce/master.go`). As tasks become available, `schedule()` — located in `mapreduce/schedule.go` — decides how to assign those tasks to workers, and how to handle worker failures.
- 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. It does so either directly (when using `Sequential()`) or by issuing the `DoTask` RPC — located in `mapreduce/worker.go` — on a worker. 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. Thus, after all map tasks are done, the total number of files will be the product of the number of files given to map (`nIn`) and `nReduce`.

```
f0-0, ..., f0-[nReduce-1],
...
f[nIn-1]-0, ..., f[nIn-1]-[nReduce-1].
```

- The master next makes a call to `doReduce()` in `mapreduce/common_reduce.go` at least once for each reduce task. As with `doMap()`, it does so either directly or through a worker. `doReduce()` collects corresponding files from each map result (e.g. `f0-i, f1-i, ... f[nIn-1]-i`), and runs the reduce function on each collection. This process produces `nReduce` result files.
- 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.
- The master sends a Shutdown RPC to each of its workers, and then shuts down its own RPC server.

You should look through the files in the MapReduce implementation, as reading them might be useful to understand how the other methods fit into the overall architecture of the system hierarchy. However, for this assignment, you will write/modify only `doMap` in `mapreduce/common_map.go`, `doReduce` in `mapreduce/common_reduce.go`, and `mapF` and `reduceF` in `main/wc.go`. You will not submit other files or modules.

2.1 Part 1: Map/Reduce input and output

The Map/Reduce implementation you are given is missing some pieces. Before you can write your first Map/Reduce function pair, you will need to fix the sequential implementation. In particular, the code 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.

To help you determine if you have correctly implemented `doMap()` and `doReduce()`, you are provided 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 top-level directory (that is, the top level of the hierarchy contained in the tarball you downloaded above):

```
$ cp -R ~swe4004/project1 ~/
$ cd ~/project1 # or whereever you copied the source code
$ export GOPATH="$PWD"
$ cd src
$ go test -run Sequential mapreduce/...
ok mapreduce 4.515s
```

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. You will get much more output along the lines of:

```
$ go test -v -run Sequential
=== RUN TestSequentialSingle
master: Starting Map/Reduce task test
Merge: read mrtmp.test-res-0
master: Map/Reduce task completed
--- PASS: TestSequentialSingle (2.30s)
=== RUN TestSequentialMany
master: Starting Map/Reduce task test
Merge: read mrtmp.test-res-0
Merge: read mrtmp.test-res-1
Merge: read mrtmp.test-res-2
master: Map/Reduce task completed
--- PASS: TestSequentialMany (2.32s)
PASS
ok mapreduce4.635s
```

2.2 Part 2: Single-worker Word Count

Now that the map and reduce tasks are connected, we can start implementing some interesting Map/Reduce operations. For this assignment, we will be implementing word count — a simple and classic Map/Reduce example. Specifically, your task is to modify `mapF` and `reduceF` within `main/wc.go` so that the application reports the number of occurrences of each word. A word is any contiguous sequence of letters, as determined by `unicode.IsLetter`.

There are some input files with pathnames of the form `pg-*.txt` in the `main` directory, downloaded from Project Gutenberg. This is the result when you initially try to compile the code we provide you and run it:

```
$ cd "$GOPATH/src/main"
$ go run wc.go master sequential pg-*.txt
# command-line-arguments
./wc.go:14: missing return at end of function
./wc.go:21: missing return at end of function
```

The compilation fails because we haven't written a complete map function (`mapF()`) nor a complete reduce function (`reduceF()`) in `wc.go` yet. Before you start coding read Section 2 of the MapReduce paper. Your `mapF()` and `reduceF()` functions will differ a bit from those in the paper's Section 2.1. 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.

You can test your solution using:

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

The output will be in the file `mrtmp.wcseq`. We will test your implementation's correctness with the following command, which should produce the following top 10 words:

```
$ 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
```

(this sample result is also found in `main/mr-testout.txt`)

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

```
$ rm mrtmp.*
```

To make testing easy for you, from the `src/main` directory, run:

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

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

2.3 Resources and Advice

- http://blog.golang.org/strings : a good read on what strings are in Go is the Go Blog on strings.

- http://golang.org/pkg/strings/#FieldsFunc : you can use `strings.FieldsFunc` to split a string into components.
- http://golang.org/pkg/strconv/ : the strconv package is handy to convert strings to integers etc.

3 Distributed MapReduce

3.1 Part 3: Distributing MapReduce tasks

Your current implementation runs all the map and reduce tasks one after another on the master. While this is conceptually simple, it is not great for performance. In this part of the assignment, you will complete a version of MapReduce that splits the work up over a set of worker threads, in order to exploit multiple cores. Computing the map tasks in parallel and then the reduce tasks can result in much faster completion, but is also harder to implement and debug. Note that for this assignment, the work is not distributed across multiple machines as in "real" Map/Reduce deployments, your implementation will be using RPC and channels to simulate a truly distributed computation.

To coordinate the parallel execution of tasks, we will use a special master thread, which hands out work to the workers and waits for them to finish. To make the assignment more realistic, the master should only 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_rpc.go`).

Your job is to complete `schedule.go` in the `mapreduce` package. In particular, you should modify `schedule()` in `schedule.go` to hand out the map and reduce tasks to workers, and return only when all the tasks have finished.

Look at `run()` in `master.go`. It calls your `schedule()` to run the map and reduce tasks, then calls `merge()` to assemble the per-reduce-task outputs into a single output file. `schedule` only needs to tell the workers the name of the original input file (`mr.files[task]`) and the task `task`; each worker knows from which files to read its input and to which files to write its output. The master tells the worker about a new task by sending it the RPC call `Worker.DoTask`, giving a `DoTaskArgs` object as the RPC argument.

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

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 (the correct functions are given to them when they are started by `main/wc.go`).

To test your solution, you should run the Go test suite as follows. This will execute the distributed test case without worker failures instead of the sequential ones we were running before:

```
$ go test -run TestBasic mapreduce/...
```

As before, you can get more verbose output for debugging if you set `debugEnabled = true` in `mapreduce/common.go`, and add `-v` to the test command above. You will get much more output along the lines of:

```
$ go test -v -run TestBasic mapreduce/...
=== RUN TestBasic
/var/tmp/824-32311/mr8665-master: Starting Map/Reduce task test
Schedule: 100 Map tasks (50 I/Os)
/var/tmp/824-32311/mr8665-worker0: given Map task #0 on file 824-mrinput-0.txt
    (nios: 50)
/var/tmp/824-32311/mr8665-worker1: given Map task #11 on file 824-mrinput-11.txt
    (nios: 50)
/var/tmp/824-32311/mr8665-worker0: Map task #0 done
/var/tmp/824-32311/mr8665-worker0: given Map task #1 on file 824-mrinput-1.txt
    (nios: 50)
```

```
/var/tmp/824-32311/mr8665-worker1: Map task #11 done
/var/tmp/824-32311/mr8665-worker1: given Map task #2 on file 824-mrinput-2.txt
    (nios: 50)
/var/tmp/824-32311/mr8665-worker0: Map task #1 done
/var/tmp/824-32311/mr8665-worker0: given Map task #3 on file 824-mrinput-3.txt
    (nios: 50)
/var/tmp/824-32311/mr8665-worker1: Map task #2 done
...
Schedule: Map phase done
Schedule: 50 Reduce tasks (100 I/Os)
/var/tmp/824-32311/mr8665-worker1: given Reduce task #49 on file
    824-mrinput-49.txt (nios: 100)
/var/tmp/824-32311/mr8665-worker0: given Reduce task #4 on file
    824-mrinput-4.txt (nios: 100)
/var/tmp/824-32311/mr8665-worker1: Reduce task #49 done
/var/tmp/824-32311/mr8665-worker1: given Reduce task #1 on file
    824-mrinput-1.txt (nios: 100)
/var/tmp/824-32311/mr8665-worker0: Reduce task #4 done
/var/tmp/824-32311/mr8665-worker0: given Reduce task #0 on file
    824-mrinput-0.txt (nios: 100)
/var/tmp/824-32311/mr8665-worker1: Reduce task #1 done
/var/tmp/824-32311/mr8665-worker1: given Reduce task #26 on file
    824-mrinput-26.txt (nios: 100)
/var/tmp/824-32311/mr8665-worker0: Reduce task #0 done
...
Schedule: Reduce phase done
Merge: read mrtmp.test-res-0
Merge: read mrtmp.test-res-1
...
Merge: read mrtmp.test-res-49
/var/tmp/824-32311/mr8665-master: Map/Reduce task completed
--- PASS: TestBasic (25.60s)
PASS
ok mapreduce25.613s
```

3.2 Part 4: 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 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.)

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 rest of this course is devoted to this challenge.

Your implementation must pass the two remaining test cases in `test_test.go`. The first case tests the failure of one worker, while 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 tasks. To run these tests:

```
$ go test -run Failure mapreduce/...
```

3.3 Part 5: Inverted index generation

Word count is a classical example of a Map/Reduce application, but it is not an application that many large consumers of Map/Reduce use. It is simply not very often you need to count the words in a really large dataset. For this application exercise, we will instead have you build Map and Reduce functions for generating an inverted index.

Inverted indices are widely used in computer science, and are particularly useful in document searching. Broadly speaking, an inverted index is a map from interesting facts about the underlying data, to the original location of that data. For example, in the context of search, it might be a map from keywords to documents that contain those words.

We have created a second binary in `main/ii.go` that is very similar to the `wc.go` you built earlier. You should modify `mapF` and `reduceF` in `main/ii.go` so that they together produce an inverted index. Running `ii.go` should output a list of tuples, one per line, in the following format:

```
$ go run ii.go master sequential pg-*.txt
$ head -n5 mrtmp.iiseq
A: 16
    pg-being_ernest.txt,pg-dorian_gray.txt,pg-dracula.txt,pg-emma.txt,pg-frankenstein.txt,pg-great
ABC: 2 pg-les_miserables.txt,pg-war_and_peace.txt
ABOUT: 2 pg-moby_dick.txt,pg-tom_sawyer.txt
ABRAHAM: 1 pg-dracula.txt
ABSOLUTE: 1 pg-les_miserables.txt
```

If it is not clear from the listing above, the format is:

```
word: #documents documents,sorted,and,separated,by,commas
```

We will test your implementation's correctness with the following command, which should produce these resulting last 10 items in the index:

```
$ sort -k1,1 mrtmp.iiseq | sort -snk2,2 mrtmp.iiseq | grep -v '16' | tail -10
women: 15
    pg-being_ernest.txt,pg-dorian_gray.txt,pg-dracula.txt,pg-emma.txt,pg-frankenstein.txt,pg-great
won: 15
    pg-being_ernest.txt,pg-dorian_gray.txt,pg-dracula.txt,pg-frankenstein.txt,pg-great_expectation
wonderful: 15
    pg-being_ernest.txt,pg-dorian_gray.txt,pg-dracula.txt,pg-emma.txt,pg-frankenstein.txt,pg-great
words: 15
    pg-dorian_gray.txt,pg-dracula.txt,pg-emma.txt,pg-frankenstein.txt,pg-great_expectations.txt,pg
worked: 15
    pg-dorian_gray.txt,pg-dracula.txt,pg-emma.txt,pg-frankenstein.txt,pg-great_expectations.txt,pg
worse: 15
    pg-being_ernest.txt,pg-dorian_gray.txt,pg-dracula.txt,pg-emma.txt,pg-frankenstein.txt,pg-great
wounded: 15
    pg-being_ernest.txt,pg-dorian_gray.txt,pg-dracula.txt,pg-emma.txt,pg-frankenstein.txt,pg-great
yes: 15
    pg-being_ernest.txt,pg-dorian_gray.txt,pg-dracula.txt,pg-emma.txt,pg-great_expectations.txt,pg
younger: 15
    pg-being_ernest.txt,pg-dorian_gray.txt,pg-dracula.txt,pg-emma.txt,pg-frankenstein.txt,pg-great
yours: 15
    pg-being_ernest.txt,pg-dorian_gray.txt,pg-dracula.txt,pg-emma.txt,pg-frankenstein.txt,pg-great
```

(this sample result is also found in main/mr-challenge.txt)

To make testing easy for you, from the `$GOPATH/src/main` directory, run:

```
$ sh ./test-ii.sh
```

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

4 Submission

5 Source Code Submission

I expect you work on In-Ui-Ye-Ji cluster, which consists of four nodes. Please submit your source code files in swin.skku.edu using distsys_submit command as follows.

```
$ distsys_submit mapreduce /your/code/directory/path
```

This command will compress and submit all files in the specified directory. For example, if your source codes are in the current directory, run the following command.

```
$ distsys_submit mapreduce ./
```

Note that you can submit multiple times. But only the last submission will be graded. Using the following command, you can check whether your file has been correctly submitted.

```
$ distsys_check_submission mapreduce
```

6 Demo Video Submission

In addition to the source code submission, you need to create a demo video to demonstrate whether Part 1, 2, 3, 4, and 5 are executed correctly. This demo video will replace the report, i.e., there is no need to write a report. Submission menu will be set up on iCampus, later.

7 Acknowledgement

This assignment is adapted from MIT's 6.824 course. Thanks to Frans Kaashoek, Robert Morris, and Nickolai Zeldovich for their support.