# Lab-2: Word Count with MapReduce

Due: 11-1-2021 23:59 (UTC+8)

## Introduction

- This lab includes 2 parts.
- In Part-1 (50 points), you are asked to build a Distributed Filesystem with a given RPC library.
- In Part-2 (50 points), you'll build a MapReduce framework upon the Distributed Filesystem that you have implemented in Part-1.
- If you have any questions, please feel free to let the TA know: Mingyu Li (maxullee@sjtu.edu.cn).
- Sincerely hope this lab can help you learn something. Enjoy the lab!

## Getting started

Please back up your lab1 solution before progressing the steps below:

In the first place, please remember to save your lab1 solution:

```
% cd cse-lab
% git commit -a -m "solution for lab1"
```

Then, pull from the repository:

```
% git pull
remote: Counting objects: 43, done.
…
[new branch]  lab2  -> origin/lab2
Already up-to-date
```

Then, switch to the lab2 branch:

```
% git checkout lab2
```

Merge with lab1, and solve conflicts on your own (mainly in `fuse.cc` and `chfs_client.cc`):

```
% git merge lab1
Auto-merging fuse.cc
CONFLICT (content): Merge conflict in chfs_client.cc
Auto-merging chfs_client.cc
CONFLICT (content): Merge conflict in ifs_client.cc
Automatic merge failed; fix conflicts and then commit the result
……
```

After merging all conflicts, you should be able to compile the new project successfully:

```
% chmod -R o+w `pwd`
% sudo docker run -it --rm --privileged --cap-add=ALL -v `pwd`:/home/stu/cse-lab
shenjiahuan/cselab_env:1.0 /bin/bash
% cd cse-lab
% make
```

Please make sure there is no error during `make`.

- **Note**: For lab2, you need not worry about server failures or client failures, nor concern about malicious or buggy applications. You'll try to address them in the next lab.

## Distributed File Server

In lab1, you have implemented a file system named `ChFS` on a single machine. In lab2, your objective is to extend it to a distributed file server with RPC.

Luckily, most of your jobs have been done in lab1. You now can use extent service provided by `extent_server` through RPC in `extent_client`.

*You'd better test your code with the previous test suit before any progress.*

### Using the RPC library

- In this lab, you don't need to care about the implementation of the underlying RPC mechanisms. Instead, you'll use an existing RPC system to make your local filesystem become a distributed filesystem.
- A server uses the RPC library by creating an RPC server object `rpcs` listening on a port and registering various RPC handlers (see `main()` function in `demo_server.cc`).
- A client creates an RPC client object `rpcc`, asks for it to be connected to the demo_server's address and port, and invokes RPC calls (see `demo_client.cc`).
- You can learn how to use the RPC library by studying the `stat` call implementation. Note that it's for the illustration purpose only, you won't need to follow the implementation.
- You can use `make rpcdemo` to build the RPC demo.

### Hints

- RPC handlers have a standard interface with 1 to 6 request arguments and 1 reply value as a last reference argument. The handler also returns an integer status code; the convention is to return zero for success and to return positive numbers for various errors.
- If the RPC fails in the RPC library (e.g.timeouts), the RPC client gets a negative return value instead. The various reasons for RPC failures in the RPC library are defined in `rpc.h` under `rpc_const`.
- The RPC marshalls objects into a stream of bytes to transmit over the network and unmarshalls them at the other end.
- Beware: the RPC library does not check that the data in an arriving message have the expected type(s). If a client sends one type and the server is expecting a different type, something bad will happen. You should check the client's RPC call function sends types that are the same as those expected by the corresponding server handler function.
- The RPC library provides marshall/unmarshall methods for standard C++ objects such as `std::string`, `int`, and `char`. You should be able to complete this lab with existing marshall/unmarshall methods.
- This is the first lab that writes null ('\0') characters to files. The `std::string(char*)` constructor treats '\0' as the end of the string, so if you use that constructor to hold file content or the written data, you will have trouble with this lab. Use the `std::string(buf, size)` constructor instead. Also, if you use C-style `char[]` carelessly you may run into trouble :(

### Grading

To grade this part of lab, a test script grade.sh is provided. Here's a successful grading.

```
% ./grade.sh
Passed A
Passed B
Passed C
Passed D
Passed E
Passed G (Consistency)
Lab2 part 1 passed
……
```

Notice that a non-RPC version may also pass the tests, but RPCs will be checked against in actual grading. So please refrain yourself from doing so :D

## MapReduce

(Reference: MIT 6.824 Distributed Systems)

In this lab, you are asked to build a MapReduce framework.

You will implement a worker process that calls Map and Reduce functions and handles reading and writing files, and a coordinator process that hands out tasks to workers and copes with failed workers.

You can refer to the MapReduce paper for more details (Note that this lab uses "coordinator" instead of the paper's "master").

There are four files added for this part: `mr_protocol.h`, `mr_sequential.cc`, `mr_coordinator.cc`, `mr_worker.cc`.

### Task 1

- `mr_sequential.cc` is a sequential mapreduce implementation, running Map and Reduce once at a time within a single process.
- You task is to implement the Mapper and Reducer for Word Count in `mr_sequential.cc`.

### Task 2

- Your task is to implement a distributed MapReduce, consisting of two programs, `mr_coordinator.cc` and `mr_worker.cc`. There will be only one coordinator process, but one or more worker processes executing concurrently.
- The workers should talk to the coordinator via the `RPC`. One way to get started is to think about the RPC protocol in `mr_protocol.h` first.
- Each worker process will ask the coordinator 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 coordinator should notice if a worker hasn't completed its task in a reasonable amount of time, and give the same task to a different worker.
- 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.
- MapReduce relies on the workers sharing a file system. This is why we ask you to implement a global distributed ChFS in the first place.

#### Hints

- The number of Mappers equals to the number of files be to processed. Each mapper only processes one file at one time.
- The number of Reducers equals is a fixed number defined in `mr_protocol.h`.
- The basic loop of one worker is the following: ask one task (Map or Reduce) from the coordinator, do the task and write the intermediate key-value into a file, then submit the task to the coordinator in order to hint a completion.
- The basic loop of the coordinator is the following: assign the Map tasks first; when all Map tasks are done, then assign the Reduce tasks; when all Reduce tasks are done, the `Done()` loop returns true indicating that all tasks are completely finished.
- Workers 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 coordinator for work, sleeping between each request. Another possibility is for the relevant RPC handler in the coordinator to have a loop that waits.
- The coordinator, as an RPC server, should be concurrent; hence please don't forget to lock the shared data.
- The Map part of your workers can use a hash function to distribute the intermediate key-values to different files intended for different Reduce tasks.
- 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.

### Grading

After you have implement part1 & part2, run the grading script:

```
% ./grade.sh
Passed part1 A
Passed part1 B
Passed part1 C
Passed part1 D
Passed part1 E
Passed part1 G (Consistency)
Lab2 part 1 passed

Passed part2 A (Word Count)
Passed part2 B (Word Count with distributed MapReduce)
Lab2 part 2 passed

Passed all tests!
Score: 100/100
```

We will test your MapReduce following the evaluation criteria above.

## Handin Procedure

After all above done:

```
% make handin
```

That should produce a file called lab2.tgz in the directory. Change the file name to your student id:

```
% mv lab2.tgz lab2_[your student id].tgz
```

Then upload **lab2_[your student id].tgz** file to Canvas before the deadline.

You'll receive full credits if your code passes the same tests that we gave you, when we run your code on our machines.